X68k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義………共著

(#1)

X680x0 Develop.

Vol. 2

Reference

Manual Books

SOFT BANK

X68k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義……共著

(#1)

X68000 Develop.

Vol. 2

Reference

本書は,「*X680x0 Develop. & libc II*」の内容に即して「*X68000 Develop.*」を加筆修正したものです。
便宜上，文章中「X680x0 ***では」と記載されているのは,「*X680x0 Develop. & libc II*」に収録されている各ツールを指し,「従来では」または「X68000 *」と記載されているものは,「*X68000 Develop.*」に収録されている各ツールを指します。**

- 本書に記載したすべてのプログラムは,「GNU 一般公有使用許諾書」を適用しています。
- その他のシステム名，CPU 名などは一般に各社の登録商標です。本文中では，とくに TM, Ⓡ は明記しておりません。

X68k Programming Series #1

X680x0 Develop.

Vol.2 Reference

C O N T E N T S

Chapter 1

オプションスイッチ 1
1.1 ―― **GCCオプション** 2
- コンパイラドライバオプション概要 2
- コンパイラドライバのオプションスイッチ 3
- X68000 GCC独自のオプションスイッチ 33
- 実行ファイルの条件を指定するオプションスイッチ 44
- プリプロセッサのオプションスイッチ 45
- コンパイラ本体のオプションスイッチ 47

1.2 ―― **アセンブラオプション** 48
1.3 ―― **リンカオプション** 68
1.4 ―― **デバッガオプション** 82

Chapter 2

診断メッセージ 99
2.1 ―― **GCC診断メッセージ** 100
- 宣言に関するエラー 100
- 初期化でのエラー 126
- 型変換, sizeof でのエラー 133
- 関数記述でのエラー 135
- GCC拡張でのエラー 153
- 宣言に関するワーニング 157
- 関数記述でのワーニング 162
- GCC拡張でのワーニング 169

2.2 ―― **アセンブラのメッセージ** 172
- アセンブルを中断するエラーメッセージ 172
- 通常のエラーメッセージ 173
- ワーニングメッセージ 174

2.3 ―― **HLKのメッセージ** 176

Chapter 3

GDBのコマンド 179
3.1 ―― **コマンドリファレンス** 180
- 実行を制御するコマンド 181
- スタックフレームを調査するコマンド 186
- データに関するコマンド 188
- ブレークポイントに関するコマンド 193
- ファイルに関するコマンド 196
- プログラムの状態を調査するコマンド 199
- シンボルテーブルを調査するコマンド 200
- info コマンドのサブコマンド 201
- GDBを設定するコマンド 205

3.2 ―― **行編集をサポートするキー** 212

Chapter 4

Appendix …… 213
4.1 ──── **各ツールオプション一覧** …… 214
- コンパイラドライバのオプションスイッチ …… 214
- X68000 GCC 独自のオプションスイッチ …… 215
- 実行ファイルの条件を指定するオプションスイッチ …… 215
- アセンブラオプション …… 216
- リンカオプション …… 216
- デバッガオプション …… 217

4.2 ──── **GDB コマンド一覧** …… 218
- 実行を制御するコマンド …… 218
- スタックフレームを調査するコマンド …… 218
- データに関するコマンド …… 219
- ブレークポイントに関するコマンド …… 219
- ファイルに関するコマンド …… 219
- プログラムの状態を調査するコマンド …… 220
- シンボルテーブルを調査するコマンド …… 220
- info コマンドのサブコマンド …… 220
- GDB を設定するコマンド …… 221

4.3 ──── **診断メッセージ一覧** …… 222
- GCC エラーメッセージ …… 222
- GCC ワーニングメッセージ …… 226
- アセンブラのメッセージ …… 229
- HLK のメッセージ …… 230

4.4 ──── **アセンブラ擬似命令一覧** …… 232
- アセンブラ制御 …… 232
- セクション指定 …… 232
- 外部名の宣言 …… 233
- シンボルの定義 …… 233
- マクロ制御 …… 233
- データの定義 …… 233
- 条件つきアセンブル …… 234
- リストファイル制御 …… 234
- シンボリックデバッグ情報の指定 …… 234

Chapter 1

オプションスイッチ

本章では，各ツールのコマンドライン上で使用するオプションスイッチについて解説します。

1.1 ……………………………… GCC オプション

GCC を構成する 3 つのプログラムのコマンドラインオプションスイッチについて説明します。**GCC** は，以下の 3 プログラムから構成されています。

- `gcc.x`
  コンパイラドライバ
- `gcc_cpp.x`
  プリプロセッサ
- `gcc_cc1.x`
  コンパイラ本体

これらのプログラムは，それぞれコマンドラインオプションをもっています。`gcc_cpp.x` と `gcc_cc1.x` の 2 つについては，個別に実行する場合を除いて特に知る必要もないのですが，今回は，これらもセクションを設けて説明しておきました。

1.1.1 コンパイラドライバオプション概要

X68000 GCC には，豊富なコンパイラ制御オプションが用意されています。**GCC** は本来，**UNIX** で標準に用意されている CC とほぼ同様の制御オプションを備えていますが，それに加えて **X68000 GCC** には別の独自の制御オプションが追加されています。このオプションは，コマンドライン上で “-” といくつかの文字を記述することで実行されます。XC では “/” をスイッチキャラクタとして使えますが， **GCC** では “-” のみです。そのため，XC および **GCC** で共通する `Makefile` [1] を記述する場合には “/” でなく “-” を用いてください。XC は “-” もスイッチキャラクタとして受け入れ可能です。また，コンパイルオプションを与える際に注意しなければならないこともあります。たとえば「最適化あり (-O)」,「アセンブラソース生成 (-S)」の 2 つのオプションを指定する場合，

1) Makefile というのは，ソースの依存関係を考慮してコンパイル作業を自動で行うツール Make のためのファイルです。

```
A>gcc -O -S test.c
A>
```

とします。コンパイラに指示を与えるオプションは，1 つの“-”に対して 1 つです。この例で，

```
A>gcc -OS test.c
A>
```

とするのは誤りです。またいくつかのオプションは，環境変数に設定しておくことで，コンパイラに特に明示指定しなくても指定されたようにふるまわせることもできます。

本セクションでは“定義”，“宣言”といった言葉がたくさんでてきます。一般に，この言葉はわりとあいまいに用いられていますが，本セクション内での説明は非常に厳密です。また各オプションの説明は，筆者が独自に自分の言葉で記述したものです。**GCC** オリジナルドキュメントの直訳ではありません。

X68000 GCC に用意されているコンパイルオプションは，以下のとおりです。大文字と小文字は区別されます。またコマンドの説明には，可能なかぎり実際に使用した例を挿入しました。

1.1.2 コンパイラドライバのオプションスイッチ

gcc.x が受け入れることができるオプションのうち，**GCC** に最初から備えられているオプションの機能には次のものがあります。

- “-a” **ブロック単位でのプロファイラ**
- “-ansi” **ANSI 違反の報告**
- “-C” **コメントを削除しない**
- “-c” **オブジェクトファイルの生成**
- “-D” **マクロの定義**
- “-E” **プリプロセッサ処理結果の出力**
- “-f” **最適化の許可／禁止**
- “-g” **ソースコードデバッグ情報の生成**
- “-I” **インクルードパスの指定**
- “-l” **ライブラリの指定**
- “-M” **ファイル依存関係の出力**
- “-MM” **ファイル依存関係の出力**
- “-mregparm” **引数をレジスタ渡しにする**
- “-mshort” **int を 16 ビットにする**
- “-m68881” **68881 用コードの生成**
- “-O” **最適化の実行**
- “-o” **出力ファイルネームの指定**
- “-p” **プロファイラコードの生成**
- “-pedantic” **ANSI に厳密に適合**

- “-Q” バーボーズモード指定
- “-S” アセンブラソースの生成
- “-traditional” 伝統的な C 言語仕様に準拠
- “-trigraph” trigraph シーケンスの認識
- “-U” マクロの削除
- “-v” コマンドラインの表示
- “-version” コンパイラのバージョン表示
- “-W” ワーニングの許可／指定されたワーニングの許可
- “-w” ワーニングの禁止

-a　　ブロック単位でのプロファイラ

書式：　-a

機能：　関数内部のブロック単位でプロファイルを行います。

解説：　関数内部のブロック単位でのプロファイルを行います。"-SX" オプションと同時には使えません。コンパイラドライバは，リンカに対してプロファイラ情報を出力するライブラリのリンクを指示します。

-ansi　ANSI 違反の報告

書式： -ansi

機能： **ANSI C** に合致しない記述を警告します。

解説： **GCC** の **ANSI C** 規格に適合しない拡張，**asm**，**inline**，**typeof** を禁止します。また，ANSI の **trigraph** シーケンスを認識します[2)]。**GCC** は，デフォルトではこれを認識しません。また "**-ansi**" オプションは，非 ANSI プログラム記述を完全に拒絶するわけでもありません。List 1-1 はその典型的な例です。

ANSI C では "キャストされた式" は左辺値ではありませんが，このオプションではエラーにはなりません。厳密に **ANSI C** を遵守するには，"**-pedantic**" オプションと一緒に指定してください。

また，"**-ansi**" オプションはマクロ `__STRICT_ANSI__` をプリプロセッサに **#define** させます。これを利用すれば，**GCC** の拡張予約語をワーニングなしでヘッダなどで使えます。

2)GCC の作者である R.M.Stallman 氏は trigraph シーケンスを酷評しています。

```
A>gcc test.c -O -S -ansi
A>gcc test.c -O -S -ansi -pedantic
test.c: In function foo:
test.c:       4: Error   : 代入に不正な左辺値があります
A:/BIN/gcc.x: Program gcc_cc1.x exit status 33.
A>
```

List 1-1 ● ANSI に適合しない例

```
int foo (int *x)
{
  (char *)x = 0;
}
```

-C　コメントを削除しない "-E"

書式：　-C

機能：　コメントを削除しないで，プリプロセッサの出力を標準出力へ書き出します。

解説：　"-E" オプションではソースファイルに記述されているコメントを削除しますが，このオプションではコメントは削除されません。

```
A>type test.c
#include <stdio.h>
#include "my_head.h"
/* コメント */
main ()
{
}

A>gcc test.c -C
# 1 "test.c"
# 1 "a:\include\stdio.h" 1
...
...
...
...
# 1 "test.c" 2

# 1 "my_head.h" 1
int foo;

# 2 "test.c" 2
/* コメント */

main ()
{
}
A>
```

-c　オブジェクトファイルの生成

書式：　-c

機能：　リンクを実行しません。

解説：　ソースファイルをリロケータブルオブジェクトにまで展開してコンパイルを終了します。"-o" オプションで出力ファイルを指示しない場合は，ソースファイルの拡張子を ".o" で置き換えたものを出力ファイルとします。

```
A>dir
 ********              A:\
     2 ファイル       **K Byte 使用中      ***K Byte 使用可能
 ファイル使用量      **K Byte 使用
test              c          60  **-**-**  **:**:**
my_head           h          12  **-**-**  **:**:**
A>gcc test.c -c
test.c:      8: Warning: コード最適化は行われていません
A>dir
 ********              A:\
     3 ファイル       **K Byte 使用中      ***K Byte 使用可能
 ファイル使用量      **K Byte 使用
test              c          60  **-**-**  **:**:**
my_head           h          12  **-**-**  **:**:**
test              o         162  **-**-**  **:**:**
```

-D <マクロ名> マクロの定義

書式： -D <マクロ名>

機能： マクロをコマンドラインで定義します。

解説： <マクロ名> を #define します。ソースファイルの先頭に，"#define <マクロ名>" を記述した場合と等価になります。

-D <マクロ名> = <値> マクロの定義

書式： -D <マクロ名> = <値>

機能： マクロをコマンドラインで定義します。

解説： <マクロ名> を <値> に #define します。このオプションを指定すると，ソースファイルの先頭に "#define <マクロ名> <値>" を記述した場合と等価になります。シェルが解釈する文字に注意してください。シェルが扱う文字をマクロに定義させる場合は，シェルに応じたエスケープが必要です。

-E　プリプロセッサ処理結果の出力

書式： -E

機能： プリプロセッサの出力を標準出力に書き出します。

解説： プリプロセッサの処理結果を標準出力に出力して終了します。プリプロセスの結果，次のコンパイル段階でエラーになる場合の原因究明に役立ちます。

```
A>type test.c
#include <stdio.h>
#include "my_head.h"

main ()
{
}


A>gcc test.c -E
# 1 "test.c"
# 1 "a:\include\stdio.h" 1
...
...
...
...
# 1 "test.c" 2

# 1 "my_head.h" 1
int foo;

# 2 "test.c" 2


main ()
{
}
A>
```

-f <文字列> 最適化の許可および禁止

書式： -f <文字列>

機能： 特定の最適化を許可および禁止します。

解説： “-f” オプションは，特定の最適化の許可を行ったり禁止したりするためのオプションです。

-fno-<文字列> は <文字列> を禁止させる形式です。つまり **X68000 GCC** では，-fno-<文字列> 形式の説明は，デフォルトで許可されている最適化を禁止していることになります。

X68000 GCC では，環境変数でいくつかのオプションをデフォルト設定にすることができます。

❖ **-fcaller-saves**

関数呼び出しで，破壊されるレジスタ[3)]に対して変数を割り付けることを許可します。通常，このような割り付けは，よりよいと思われる場合以外は避けられます。これらのレジスタは関数呼び出しで破壊されるために，コンパイラはそれらを保存するためのコードをよぶんに生成します。68000 の場合には，レジスタが 16 本とかなり多いので，このオプションを指定しなくても，たいていは十分に速いコードを生成します。

3)X68000 GCC では d0, d1, d2, a0, a1, a2 が該当します。

❖ **-fcombine-regs**

GCC には，コンパイル時に複数の命令を単独の命令に最適化する機能があります。このオプションは，その過程においてレジスタ間の複写命令の複合化を許可します。ですが 68000 において，このオプションの使用によるオブジェクトコードの違いを筆者は発見したことがありません。

❖ **-fforce-addr**

ポインタなどでアドレスを指定してメモリ演算を行うとき，そのアドレスをレジスタに保持するようなコードを生成します。68000 ではポインタを多用した場合，レジスタアロケーションでアドレスレジスタが不足する傾向がありますので，このオプションを指定しないほうがしばしば速いコードを生成します。

❖ **-fforce-mem**

メモリ間の演算をレジスタ上で行うようなコードを生成します。**X68000 GCC** では，被演算数が定数である乗算はすべてシフトと加減算で計算するので，配列を多用している場合，このオプションを指定しておくと非常に有効です。

❖ **-finline-functions**

コンパイラは “**inline** 展開できる” 関数については，その関数を呼び出すコードを生成しないで，直接その関数の処理を呼び出し位置に埋め込みます。**inline** にするかどうかは，コンパイラ

が自分で判断します。もし，その“inline 展開できる”関数が static であれば，その関数に対応するアセンブラコードは生成されなくなります。

❖ **-fkeep-inline-functions**

“inline 展開できる”関数が static であり，すべての呼び出しが inline に展開された場合でも，その関数に対応するアセンブラコードを生成します。

❖ **-fno-defer-pop**

通常 **GCC** は，関数呼び出し後のスタック補正を複数の関数呼び出し後に一括して行います。このオプションはそれを禁止します。**X68000** では，スーパーバイザとユーザとを切り替える関数を使った場合にこのオプションが必須でした。しかし **X68000 GCC** では，環境変数の設定によって特に指示しなくても，コンパイラが認識して適切に扱うコードを生成します。

❖ **-fno-function-cse**

レジスタ間接呼び出しが可能で，かつそのほうがより速いマシンでは，関数のアドレスをレジスタに保持して，レジスタ間接呼び出しを行います。68000 はその種類の CPU なので，デフォルトで **-function-cse** オプションは許可されています。これは，それを禁止するためのオプションです。

❖ **-fno-peep-hole**

コンパイルの最終段階で行われる覗き穴最適化を禁止します。特に意味のあるオプションとは思えません。

❖ **-fomit-frame-pointer**

必要がない場合，**GCC** はスタックフレームを生成しません。68000 CPU において，たいていの関数はほとんどの変数がレジスタに割り当て可能ですので，このオプションは非常に有効です。ただし“**-g**”オプションと併用した場合には，デバッグ作業に支障をきたすことがあります。なぜならばデバッガでは，各関数はすべてスタックフレームを生成する前提になっているからです。

❖ **-fpcc-struct-return**

GCC では構造体を返す関数は，呼び出し側で返す構造体をスタック上に確保して，そのポインタを呼び出し関数に渡すことで戻り値を得ています。XC では，このような関数は static なメモリを暗黙に生成して，そこに戻り値を返してきます。したがって，通常 XC と **GCC** とでは構造体を返す関数において互換性がまったくありません。このオプションを指定すると，**GCC** は XC と互換性を保つ方法で構造体を返す関数を扱います。XC のほうは小さな構造体では効率が悪く，またリエントラントではありません。

❖ **-fstrength-reduce**

loop 内部での演算強度の軽減を行おうとします。かなりの効果が期待できますが，コンパイル速度は低下します。通常は，環境変数で設定しておきます。

❖ **-funsigned-char**

符号指定のない **char** を **unsigned char** として扱います。この **X68000 GCC** では，符号指定のない **char** を **signed char** として扱うようにして作成されています。

❖ **-fwritable-strings**

すべての文字列リテラルを個別に扱い，また実体を **.data** セクションに出力するため，文字列リテラルが書き換え可能になります。このオプションを指定しないと，正しく動作しないようなソースコードは，極めて移植性が低いといえます。

❖ **-f レジスタ割り付けオプション**

GCC には，コンパイラのレジスタの使い方について制限や指定を行うオプションが用意されています。これらのオプションはアセンブラ主体のプログラム開発には便利ですが，コンパイラのレジスタ使用方法について熟知していないと，大変な混乱を引き起こす原因にもなります。

- **-ffixed-<レジスタ名>**

  <レジスタ名> で指定されたレジスタを使わないコードを生成します。どのレジスタを使わないようにできるのかは，コンパイラの動作条件によって変化します。**X68000 GCC** において，どのような場合にも固定できるレジスタは **d2**，**d3**，**d4**，**d5**，**d6**，**d7**，**a2**，**a3**，**a4** の 9 個です。むやみにたくさんのレジスタを固定させると，コンパイラはレジスタ割り付けに失敗してアボートすることがあります。

- **-fcall-used-<レジスタ名>**

  <レジスタ名> で指定されたレジスタを揮発レジスタ，つまり関数呼び出しによって破壊されるレジスタとして扱います。

- **-fcall-saved-<レジスタ名>**

  <レジスタ名> で指定されたレジスタを，関数呼び出しでは破壊されないレジスタとして扱います。

-g　ソースコードデバッグ情報の生成

書式：　-g

機能：　**GDB** でソースコードデバッグできるようにシンボリックデバッグ情報を付加します。

解説：　**GDB** 対応フォーマットで，シンボリックデバッグ情報を付加してコンパイルします。このフォーマットは **SHARP** 純正のソースコードデバッガ SCD.X とほぼコンパチブルです。

-I <パス名> インクルードパスの指定

書式： -I <パス名>

機能： インクルードサーチパスの指定をします。

解説： 標準のインクルードパスに加えて，インクルードファイルをサーチするパスを指示します。

```
A>dir
 ********             A:\
     3 ファイル        *K Byte 使用中     ****K Byte 使用可能
 ファイル使用量       *K Byte 使用
INC                     <dir>      **-**-**  **:**:**
test               c           36  **-**-**  **:**:**
A>dir INC
 ********             A:\INC
     1 ファイル        *K Byte 使用中     ****K Byte 使用可能
 ファイル使用量       *K Byte 使用
my_head            h            2  **-**-**  **:**:**
A>type test.c
#include "my_head.h"
main()
{
}

A>gcc test.c -S -O
test.c:      1: Error :my_head.h: ファイルまたはディレクトリが
見つかりません

A:/BIN2/gcc.x: Program gcc_cpp.x exit status 33.
A>gcc test.c -S -O -IINC
A>
```

-l <ライブラリ名>　ライブラリの指定

書式：　-l <ライブラリ名>

機能：　ライブラリのリンクを指定します。

解説：　コンパイラにライブラリのリンクを指示します。"-l" オプションの後ろの文字列 <ライブラリ名> は，次の規則でライブラリファイルネームの生成と検索に用いられます。

1. <ライブラリ名> に文字列 "lib" を連結します。
2. 環境変数 "GCC_LIB" に指定されている文字列があれば，それをさらに連結しますが，何も指定されていない場合は ".a" を連結します。環境変数 "GCC_LIB" の文字列がファイルネームとして不正であっても，チェックは一切行われません。
3. 生成されたファイルネームのライブラリがカレントディレクトリに見つかればそれをそのままリンカに指定し，見つからない場合は環境変数 "lib" が示すディレクトリを検索します。検索の結果，ライブラリが存在すればそのフルパスリストを生成してリンカに指定します。しかしここでも見つからなかった場合は，カレントディレクトリにあると仮定してリンカに指示します。

-M　ファイル依存関係の出力

書式：　-M

機能：　ソースファイルの依存関係を出力します。

解説：　Makefile のひな形として，各ソースファイルの依存関係を標準出力に書き出すようプリプロセッサに指示します。"-M" オプションでは，"#include" されたファイルすべて (system ヘッダも含めて) についての依存関係を出力します。

-MM　ファイル依存関係の出力

書式： -MM

機能： ソースファイルの依存関係を出力します。

解説： Makefile のひな形として，各ソースファイルの依存関係を標準出力に書き出します。しかし“-MM”オプションでは，“#include”されたファイルについての依存関係のみ出力し，system ヘッダについては出力しません。

-mregparm　引数をレジスタ渡しにする

書式： -mregparm

機能： 関数呼び出しをレジスタ渡しで行います。

解説： 関数の引数を，レジスタ経由で渡すようなコードを生成します。XC のライブラリはすべてスタック経由です。このオプションを指定した場合には，XC のライブラリは使用できません。なお，オリジナル **GCC** の 68000 バージョンではこのオプションの指定は動作保証されていません。**X68000 GCC** では，**GCC** 自体のコンパイルが可能な程度の安全性は確保されています。

-mshort　int を 16 ビットにする

書式：　`-mshort`

機能：　int を 16 ビットにします。

解説：　“int” を 16 ビット としてコードを生成します。XC のヘッダをそのまま使うことは即暴走につながります。このオプションを指定したときには，ライブラリ関数の引数が int のものはすべて long にしなければいけません。また，fprintf() のような引数の数が不定の関数に int を渡す場合は，必ず long にキャストしなければなりません。

-m68881　68881 用コードの生成

書式：　`-m68881`

機能：　float，double 計算で 68881 を使います。

解説：　浮動小数点コプロセッサ 68881 を用いたコードを生成します。もちろん，CPU が 68000 の **X68000** では 68881 を直接使うことはできません。これはフリーウェア **fppp.x** 用のオプションです。このソフトを使うと，拡張ボードの 68881 を直接駆動するコードに 68881 命令を展開します。“-m68881” オプション が指定されると，コンパイラドライバはアセンブラの起動前に fppp.x を起動しようとします。ですから，fppp.x がない場合はコンパイルに失敗します。

追記：　**X680x0 GCC** では **X68030** にコプロセッサを付加してある場合は，68881/68882 を直接駆動できます。

-O　最適化の実行

書式：　-O

機能：　出力コードを最適化します。

解説：　コンパイラに出力コードの最適化[4]を指示します。このオプションの指定によって，コンパイラはできるだけ短く速いコードを生成するようになります。また“-O”オプションを指定していない場合は，コンパイルの最後でワーニングが発生します。いくつかの有用なワーニングを出力させるためには，このオプションが必須です。

4)“最適化”は一般的用語です。GCCが必ず最適な，最も速く短いコードを生成するということではありません。

-o <ファイル名>　出力ファイルネームの指定

書式：　-o <ファイル名>

機能：　出力ファイルネームを指定します。

解説：　コンパイル結果が実行ファイル，リロケータブルオブジェクト，アセンブラソースのどれに指示されていても，同じようにコンパイル結果を <ファイル名> に出力します。

```
A>dir
 ********              A:\
     1 ファイル        *K Byte 使用中     ****K Byte 使用可能
 ファイル使用量        *K Byte 使用
test               c          6  **-**-**  **:**:**
A>gcc test.c -o foo.x -O
A>dir
 ********              A:\
     2 ファイル        *K Byte 使用中     ****K Byte 使用可能
 ファイル使用量        *K Byte 使用
test               c          6  **-**-**  **:**:**
foo                x         64  **-**-**  **:**:**
A>gcc test.c -O -S -o bar.s
A>dir
 ********              A:\
     3 ファイル        *K Byte 使用中     ****K Byte 使用可能
 ファイル使用量        *K Byte 使用
test               c          6  **-**-**  **:**:**
bar                s         40  **-**-**  **:**:**
foo                x         64  **-**-**  **:**:**
```

-p　　プロファイラコードの生成

書式：　-p

機能：　プロファイラコードを付加します。

解説：　関数単位でプロファイルを行います。“-SX” オプションとは同時に使えません。なぜならば，単にプロファイラライブラリがこのモードに対応していないためです。

-pedantic　ANSI に厳密に適合

書式：　-pedantic

機能：　**ANSI C** に厳格に準拠します。

解説：　ANSI 適合性を厳密にチェックします。“-ansi” オプションを指定したときにワーニングになっていた部分は，エラーになります[5)]。

5)GCC の作者である R.M.Stallman 氏はこのオプションは“使う意味がない”とまで記述しています。

-Q　バーボーズモード指定

書式： -Q

機能： コンパイラをバーボーズモードにします[6]。

解説： プリプロセッサ，およびコンパイラにバーボーズモードを指示します。このオプションを指定すると，たくさんのよけいなメッセージが標準エラーに書き出されます。コンパイルが待ち切れないほど巨大なソースファイルをコンパイルするときに指示しておけば，“暇つぶし”くらいにはなるでしょう。

6)“お喋りモード”のことです。UNIXのツールは寡黙で普通に起動した場合には，コンソールには何も出力しません。

-S　　アセンブラソースの生成

書式： -S

機能： アセンブラソースを生成します。

解説： ソースファイルをアセンブラソースにまで展開したところでコンパイルを中断します。“-o” オプションで出力ファイルを指示しない場合は，ソースファイルの拡張子を “.s” で置き換えたものを出力ファイルとします。

```
A>dir
 ********               A:\
     2 ファイル      **K Byte 使用中      ***K Byte 使用可能
 ファイル使用量       **K Byte 使用
test              c          60  **-**-**  **:**:**
my_head           h          12  **-**-**  **:**:**
A>gcc test.c -S
test.c:      8: Warning: コード最適化は行われていません
A>dir
 ********               A:\
     3 ファイル      **K Byte 使用中      ***K Byte 使用可能
 ファイル使用量       **K Byte 使用
test              c          60  **-**-**  **:**:**
my_head           h          12  **-**-**  **:**:**
test              s         162  **-**-**  **:**:**
```

-traditional　伝統的な C 言語仕様に準拠

書式：　-traditional

機能：　古い形式の C 言語に準拠します。

解説：　古い形式の C に準拠します。おもな変化は次のとおりです。

- 関数内部で宣言された extern は，そのファイル全体に有効です。つまり XC と同じになります。**ANSI C** では，ブロック内部の extern 宣言はそのブロック内部でのみ有効です。
- unsigned short，unsigned char は unsigned int に変換されます。
- マクロ __STDC__ は #define されなくなります。
- 文字列内マクロ展開が有効になります。

-trigraph　trigraph シーケンスの認識

書式：　-trigraph

機能：　トリグラフシーケンスを認識します。

解説：　-trigraph シーケンスを認識します。

-U <マクロ名>　マクロの削除

書式： -U <マクロ名>

機能： 指定されたマクロを削除します。

解説： <マクロ名> を **#undef** します。ソースファイルの先頭に，“**#undef** <マクロ名>” を記述した場合と等価になります。

-v　コマンドラインの表示

書式： -v

機能： コマンドラインをエコーします。

解説： コンパイラドライバが起動するプログラムとそのコマンドラインを標準エラー出力に書き出します。メモリ不足などで **gcc.x** を経由できない場合に，バッチファイルでコンパイルを行うときに便利です。

-version　コンパイラのバージョン表示

書式：　-version

機能：　コンパイラのバージョンを表示します。

解説：　gcc.x のバージョンナンバーを表示します。

-W　ワーニングの許可

書式： -W

機能： いくつかのワーニングを許可します。

解説： いくつかの有用なワーニングを許可します。ワーニングは有用ですが，場合によっては意味のないワーニングが発生することがありますので十分に注意してください。

- **未初期化変数の使用に対するワーニング**

  local 変数で，初期化あるいは代入される前にその値を使われた場合に警告します。ただしこのワーニングは“-O”オプションを指定しないと出力しません。なぜならば“-O”オプションによって，コンパイラは関数内の変数の寿命や使われ方を検査し[7]，その情報をもとにワーニングを発生させるからです。また，このワーニングは「レジスタを変数として使える」場合にしか発生しません[8]。ですから，volatile と宣言された変数あるいは配列，union，struct 等[9]では，このワーニングは発生しません。また GCC は union，struct も場合によってはレジスタに割り当てますが，この場合でもワーニングは発生しません。

  プログラムの流れによって必ず初期化される変数でも，このワーニングが発生することがあります。List 1-2 で foo() 関数の引数が 0 か 1 にかぎられる場合には，ワーニングは意味がありません。ですが，一般的にはこのようなプログラミングは危険です。

```
A>gcc test.c -O -W -S
test.c: In function foo:
test.c:        3: Warning: ‘b’が初期化されずに使用されているようです
A>
```

List 1-2 ● 未初期化変数のワーニング

```
foo (int a)
{
  int b;
  if (a == 0)
    b = 0;
  else if (a == 1)
    b = -1;
  bar (b);
}
```

7) これを flow-analyze と呼びます。

8) 宣言の register キーワードとは無関係です。

9) union，struct へのポインタ変数ではワーニングが発生します。

- **ライブラリ関数 longjmp による auto 変数の破壊**

  ライブラリ関数 **setjmp**, **longjmp** は C 言語に non-local な **goto** を提供するライブラリですが，不用意な使用は予想外の結果を招くことがあります。List 1-3 はかなり極端な例ですが，そのままコンパイルすると次のようになります。**GCC** は，このような呼び出しを発見して警告します。このワーニングも "-O" を指定しないと発生しませんので，注意してください。

```
A>gcc test.c -O -W -S
test.c: In function foo:
test.c:         11: Warning: 'x' は'longjmp' で破壊される可能
性があります
A>
```

List 1-3 ● setjmp による変数の破壊

```
#include <setjmp.h>
extern int bar0 (void);
void bar1 (int);
jmp_buf regs;

void
foo (void)
{
  int x = bar0 ();
  if (setjmp (regs))
    {
      x = 0;
    }
  bar1 (x);
}
```

- **関数の戻り値のチェック**

  GCC は，非 **void** 型関数が明示的に **return** で値を呼び出し，関数に返す場合とそうでない場合との両方が含まれている List 1-4 のようなときに警告します。

  また **GCC** は **exit()** 関数や **abort()** 関数のように，その関数に復帰しない関数を知りませんので，記述によってはみかけ上のワーニングが出力される場合があります。

```
A>gcc test.c -W -S
test.c: In function foo:
test.c:         6: Warning: 関数が値を返す場合，そうでない場
合があります
A>
```

List 1-4 ● 関数の戻り値検査

```
int
foo (int x)
{
 if (x == 0)
   return 0;
}
```

- **副作用のない文**

  “文” が副作用をもたない場合に警告します。List 1-5 は極端な例です。

  普通のプログラムでは滅多に発生しませんが，複雑なマクロ展開をプリプロセッサで行ったりすると，プログラマが予想していない展開結果が，このワーニングを引き起こすことがあります。

```
A>gcc test.c -O -S -W
test.c: In function foo:
test.c:     5: Warning: 意味のない文です
A>
```

List 1-5 ● 意味のない文

```
int a,b,c;
void
foo (void)
{
 a,b,c;
}
```

-W <文字列> 指定されたワーニングの許可

書式： -W <文字列>

機能： 指定されたワーニングを許可します。

解説： <文字列> に対応するワーニングを出します。<文字列> には次のものが指示できます。

- **all**
  "-W" オプション単独で出力されるワーニングを含む，ここまでの "-W" オプションで行われるワーニングのすべてを許可します。
- **cast-qual**
  "型修飾子"[10]を指定したポインタを，指定のない同じタイプにキャストした場合にワーニングが発生します。
- **comment**
  "/*" を発見した後，さらに "/*" を見つけるとワーニングが発生します。このようなことはコメントのネストに現れますが，GCC のプリプロセッサはコメントのネストを正しく処理できません。ですから通常は，このワーニングの出た後は必ずエラーになります。
- **id-clash-<定数>**
  識別子の分別長を <定数> バイトに制限した場合，制限長を超えて同じ文字が識別子に使われたときにワーニングが発生します。
- **implicit**
  暗黙の関数宣言，つまり今まで何の宣言もなく[11]関数が使われた場合にワーニングが発生します。
- **return-type**
  戻り値の型を指定しないで関数を定義すると，デフォルトで int を返す関数の宣言と同じように扱われます。このオプションをつけると，そのような暗黙な宣言に出会うたびにワーニングを出します。またその定義する関数で，明示的に return で値を返していない場合もワーニングが発生します。
- **shadow**
  スコープが異なる同一の識別子が使われた場合，スコープから外の同一名称識別子が見えなくなる場合にワーニングが発生します。
- **unused**
  ローカルな変数[12]が使われなかった場合，static と宣言した関数が定義されなかった場合にワーニングが発生します。
- **write-strings**
  GCC では文字列リテラルは書き換え不可能です。以下の例では，関数の引数は const char * に変換されます。

10)type-qualifier です。X68000 GCC では remote, common などが拡張されていますが，これらは "記憶クラス指定子" として実装されています。ですから volatile, const しか type-qualifier はありません。

11)宣言の形式はプロトタイプでなくても OK です。また，関数の定義は宣言と等価です。

12)関数内において static, extern でない宣言をした変数です。

次のプログラム (List 1-6) の場合に，**getenv()** 関数が厳密に **getenv(const char *)** と宣言されていないとワーニングが発生します。このオプションは，プログラムが不正な文字列リテラルの書き換えを防ぐためには有効です。特に SX-Window プログラムでは，リエントラント性を失わないためにもこのような行為は未然に防ぐべきですが，十分慎重に，宣言とプロトタイプを行わないと非常にたくさんのワーニングメッセージに悩まされるだけです。

XC Ver. 1，XC Ver. 2 のヘッダでは **const**，**volatile** がサポートされていないために，これらの適切な宣言はなされていません。

List 1-6 ● 文字列ポインタの変更

```
main ()
{
  char *const env = getenv ("ENVVALUE");
}
```

-w　ワーニングの禁止

書式：　-w

機能：　すべてのワーニングを禁止します。

解説：　ワーニングメッセージの出力を禁止します。よほどの理由がないかぎりは，使わないほうが身のためです。

1.1.3 X68000 GCC 独自のオプションスイッチ

次のオプションは，**X68000 GCC** において新しく独自に追加された機能を制御するものです。これらのオプションは，**X68000 GCC** を利用しているユーザからの要望および筆者の技術的興味から拡張追加されたものです。

- "-as-symbols" **アセンブラの最大シンボル数の指定**
- "-cc1-stack" **コンパイラスタック量の指定**
- "-cpp-stack" **プリプロセッサスタック量の指定**
- "-fall-bsr" **すべての関数をショートコールで指定**[13]
- "-fall-jsr" **すべての関数をロングコールで指定**[14]
- "-fall-remote" **記憶クラスの固定化**
- "-fall-text" **テキストセクションですべてを出力する**
- "-fno-const-mult-expand" **定数乗法展開の禁止**
- "-frtl-debug" **rtl の書き出し**
- "-fscd" **ソースコードデバッグの生成**
- "-fstrings-align" **文字列の偶数整合**
- "-fstack-check" **スタックチェックコードの生成**
- "-fstrings-nopcr" **文字列のプログラム相対禁止**[15]
- "-fstruct-strict-align" **構造体のパッキング**
- "-ftext-report" **詳細なエラー報告**
- "-fundump" **undump コンパイルの指定**
- "-SX" **SX–Window プログラムモードの指定**

13) X680x0 GCC では，変更されています。

14) X680x0 GCC では，廃止されています。

15) X680x0 GCC では，廃止されています。

-as-symbols=<定数> (10 進)　アセンブラの最大シンボル数の指定

書式：　-as-symbols=<定数>

機能：　アセンブラでの最大シンボル数を指定します。

解説：　アセンブラの扱えるシンボルを <定数> に指定します。**UNIX** でのフリーソフトウェア等で，アセンブラがシンボルバッファを使いつくすときがあります。そのようなとき使用してエラーを回避します。

追記：　**X680x0 HAS** を使用する場合，実質上，意味がないオプションスイッチです。

-cc1-stack=<サイズ> (10 進)　コンパイラスタック量の指定

書式：　-cc1-stack=<サイズ>

機能：　コンパイラが使うスタックサイズを指定します。

解説：　gcc_cc1.x のためのスタックサイズを <サイズ> にします。コンパイラがスタック不足でエラーになった場合に指定します。コンパイラは最適化のために大量のスタックを消費することがあります。これは関数の大きさとその中に記述されている式の量に依存します。スタックを増やした分だけ，逆に作業メモリは減少します。最適なスタック量を算定するのは非常に困難です。

-cpp-stack=<サイズ> (10進) プリプロセッサスタック量の指定

書式： -cpp-stack=<サイズ>

機能： プリプロセッサが使うスタックサイズを指定します。

解説： gcc_cpp.x のためのスタックサイズを <サイズ> にします。プリプロセッサがスタック不足でエラーになった場合に指定します。プリプロセッサはファイルをスタックに読み出すことがあります。

-fall-bsr すべての関数をショートコールで指定

書式： -fall-bsr

機能： すべての関数呼び出しを相対呼び出しにします。

解説： 小さなプログラムを作成する場合には，すべての関数呼び出しを bsr で行うことで，より小さな実行形式を得ることができます。そのような場合にこのオプションを利用します。

変更： **X680x0 GCC** では，このオプションスイッチ指定と新しい予約語 pcr を用いて完全にリロケータブルコードを生成できます。

-fall-jsr すべての関数をロングコールで指定

書式：　-fall-jsr

機能：　すべての関数呼び出しを絶対呼び出しにします。

解説：　**X68000 GCC** では，関数呼び出しは条件を満たせば bsr で行えます。しかし，**UNIX** でよく見られる非常に巨大なソースコードでは，bsr の飛び先が許容範囲を超えることがあります。このオプションはあらゆる関数をすべて bsr で呼び出すので，そのようなエラーを回避することができます。

変更：　**X680x0 GCC** では，廃止されています。

-fall-remote 記憶クラスの固定化

書式：　-fall-remote

機能：　SX モードですべての変数を remote にします。

解説：　SX–Window プログラムモードで，通常に宣言された変数をすべて remote 宣言と等価にします。変数のトータルが 64K バイトを超える場合に指定すればリンクエラーは回避できますが，本来はきめ細かく remote 宣言で対処すべき問題です。

-fall-text　テキストセクションですべてを出力する

書式：　`-fall-text`

機能：　全ソースファイルをテキストセクションとしてコンパイルします。

解説：　常駐プログラムの一部を C で記述する場合に，変数エリアがデータセクションにあると不便です。このオプションを指定すると，コンパイラは全変数を `.text` セクションに出力します。またこのオプションを指定した場合には，変数の宣言に制限が生じます。さらに，SX-Window プログラムモードではこのオプションは指定できません。

-fno-const-mult-expand　定数乗法展開の禁止

書式：　`-fno-const-mult-expand`

機能：　乗法の定数展開を禁止します。

解説：　定数との乗算をシフト／加減算に展開しません。通常の **GCC** と同じように，`gnulib` の中にある乗算サブルーチンを呼び出すコードを生成します。

-frtl-debug　rtl の書き出し

書式： `-frtl-debug`

機能： アセンブラソースに `rtl` コードをダンプします。

解説： 出力アセンブラファイルに `rtl` をダンプします。**GCC** をデバッグするために筆者が追加した機能で，通常は使うことはありません。
このオプションを指定してコンパイルしているときに，コンパイラがバスエラー等で停止したり，あるいは INTERRUPT スイッチで強制的にコンパイルを中断したりすると，現在コンパイル中の rtl を画面に出力します。

-fscd　ソースコードデバッグの生成

書式： `-fscd`

機能： ソースコードデバッグ情報を付加します。

解説： "**-g**" オプションと同じ機能です。これはかなり古いバージョンの **X68000 GCC** との互換性のためにだけ残されています。ですから，このオプションを **Makefile** 等に記述しないでください。

-fstrings-align　文字列の偶数整合

書式：　-fstrings-align

機能：　文字列リテラルを偶数整合します。

解説：　通常，文字列リテラルは偶数境界整合は行いません。そのため，MS-DOS などの境界整合の緩い CPU マシンで“違反スレスレ”で記述されたソースファイルをコンパイル／実行する場合，アドレスエラーで実行できないことがあります。このオプションを指定すると，文字列の先頭アドレスで必ず偶数整合を行いますので，上述のソースファイルを正しく実行できる場合があります。

-fstack-check　スタックチェックコードの生成

書式：　-fstack-check

機能：　スタックチェックコードを生成します。

解説：　関数の入り口で，スタックフレームを作成する前にスタック残量をチェックするコードを生成します。

-fstrings-nopcr　文字列のプログラム相対禁止

書式： -fstrings-nopcr

機能： 文字列のプログラム相対アクセスを禁止します。

解説： **X68000 GCC** では，文字列リテラルのアドレスはプログラムカウンタ相対アドレッシングをおもに利用します。これも巨大なソースコードでは，アセンブルエラーになることがあります。このオプションは，そのような場合にプログラム相対アドレス形式の利用を禁止するために使います。

変更： **X680x0 GCC** では，廃止されています。

-fstruct-strict-align 構造体のパッキング

書式： -fstruct-strict-align

機能： 構造体を厳密にパッキングします。

解説： 構造体を，構造体のメンバ境界整合条件に厳密にパッキングします。List 1-7 では，**sizeof(buf)** は通常 40 で，**buf[0]** と **buf[1]** との間には 1 バイトのすき間が存在します。このオプションを指定してコンパイルを行った場合には，**sizeof(buf)** は 30 で，**buf** 配列にはすき間は存在しません。通常アクセスでは表面上何も違いがありませんが，キャストしてコンパイラをだまして使用した場合には，当然アドレスエラーが起こる可能性があります。なお，このオプションを指定すると，XC との互換性はなくなります。

List 1-7 ● 構造体の穴

```
typedef struct {
  char red;
  char green;
  char blue;
} PALET;

PALET buf[10];
```

-ftext-report　詳細なエラー報告

書式：　-ftext-report

機能：　詳細にエラーを報告します。

解説：　ワーニング，エラーのときはソースファイルの表示を行います。このソースファイル表示機能は，コンパイラが構文解析と意味解析を行っている間だけ有効です。そのため最適化パスの後に発見されるエラーやワーニングに対しては，ソースファイルの表示は行われません。

-fundump-<ダンプファイル名>　undump コンパイルの指定

書式：　-fundump-<ダンプファイル名>

機能：　アンダンプコンパイルを行います。

解説：　コンパイルダンプ <ダンプファイル名> を読み込んでコンパイルします。

-SX　SX-Window プログラムモードの指定

書式： `-SX`

機能： SX–Window モードでコンパイルします。

解説： SX–Window プログラム開発モードにコンパイラを切り替えます。このオプションが指定されると，次のコンパイルオプションは使用できなくなります。

- `-p`
- `-a`
- `-fall-text`

これらのオプションは，“`-SX`” オプションと同時に指定するとエラーになります。

1.1.4 実行ファイルの条件を指定するオプションスイッチ

以下のオプションは，生成される実行ファイルの条件を指定するものです。このオプションは HLK.X に依存しているので，**SHARP** 純正のリンカを使う場合は指定することができません。指定するとエラーになります。

- "-z-heap" **ヒープサイズの指定**
- "-z-stack" **スタックサイズの指定**

-z-heap=<サイズ> (10 進) ヒープサイズの指定

書式： -z-heap=<サイズ>

機能： 生成実行ファイルのヒープサイズを指定します。

解説： 生成実行ファイルのヒープサイズを <サイズ> にして，実行ファイルを生成します。

-z-stack=<サイズ> (10 進) スタックサイズの指定

書式： -z-heap=<サイズ>

機能： 生成実行ファイルのスタックサイズを指定します。

解説： 生成実行ファイルのスタックサイズを <サイズ> にして，実行ファイルを生成します。

1.1.5 プリプロセッサのオプションスイッチ

プリプロセッサは独立して，テキスト処理プログラムとして使われることがあります。GNU Emacs の Makefile は，このプリプロセッサを使って作成されるファイルの一例です。プリプロセッサのオプションには，コンパイラドライバが使わない種類のオプションもあります。

❖ -i <ファイル名>

主テキスト処理を行う前に，<ファイル名> で指定されたファイルに定義されているマクロ等を読み込みます。<ファイル名> で指定されたファイルの内容は，結果には出力されません。つまり <ファイル名> で指定されたファイルの中で定義されたマクロだけが，結果に反映します。

❖ -o <ファイル名>

処理結果を <ファイル名> に出力します。この処理結果は，**Human68k** の通常のテキストファイルと異なっていて，行末改行コードが 0x0a だけになっています。

<ファイル名> に "-" を指定すると，標準出力に結果を書き出します。結果が標準出力に書き出される場合は，リダイレクトの有無に関係なく，通常の **Human68k** のテキストファイル形式になります。

❖ -p

ANSI 規約に厳密に従います。コンパイラドライバでの "-ansi" オプションと同じ意味をもちます。

❖ -traditional

伝統的なプリプロセッサの処理を行います。ANSI で新設された "#" 演算子や，"##" 演算子は認識しません。ただしマクロの展開方法は，このフラグに影響を受けないので，古い伝統的なプリプロセッサの処理結果と完全には同一にはなりません。

❖ -trigraphs

trigraph シーケンスを認識します。デフォルトでは，GNU プリプロセッサはこれを認識しません。

❖ -+

C++のコメント形式を認識します。デフォルトでは，C++形式のコメントは認識しません。

❖ -w

ワーニングメッセージの出力を禁止します。

❖ -W <文字列>

<文字列> で指定されるワーニングを許可します。<文字列> には，以下のものがあります。

- comment

  "/*" を見つけた後，さらに "/*" を見つけるとワーニングが発生しま

す。これはコメントのネストに現れますが，プリプロセッサはコメントのネストを正しく処理できません。

- **trigraphs**
  “**trigpaph**” シーケンスに出会うつど，ワーニングが発生します。

❖ **-M**

プリプロセッサに，**Makefile** のひな形になるインクルードファイルの依存関係を標準出力に書き出します。“**-M**” オプションでは，“**<**” と “**>**” で囲まれた **system** ヘッダも依存関係ファイルに出力します。

❖ **-MM**

プリプロセッサに，**Makefile** のひな形になるインクルードファイルの依存関係を標準出力に書き出しますが，“**-MM**” オプションでは，“**<**” と “**>**” で囲まれた **system** ヘッダは依存関係ファイルに出力しません。

❖ **-D <マクロ名>=<値>**

<マクロ名> を **<値>** に **#define** します。ソースファイルの先頭に，“**#define <マクロ名> <値>**” を記述したものと等価です。

❖ **-U <マクロ名>**

<マクロ名> を **#undef** します。ソースファイルの先頭に “**#undef <マクロ名>**” を記述したものと等価です。

❖ **-C**

処理されるテキスト内のコメントは削除されません。

❖ **-P**

ソースファイルの行番号情報を出力しません。

❖ **-$undef**

“**$**” を識別子として認識します。

❖ **-I <パス名>**

標準のインクルードパスに加えて，インクルードファイルをサーチするパスを指示します。

❖ **-n**

標準のインクルードパスを検索しません。カレントディレクトリ以外を捜すためには，“**-I**” オプションを同時に用いる必要があります。

❖ **-u**

あらかじめ定義されているマクロを無効にします。

❖ **-f**

処理ソースファイル内に存在する **#include** 命令で読まれたファイルは，結果を出力せずに，マクロだけを有効にします。**X68000** 版だけに存在するオプションです。

1.1.6 コンパイラ本体のオプションスイッチ

コンパイラ本体 gcc_cc1.x が受けつけるオプションは，すべてコンパイラドライバから指定できます。通常，ユーザが意識する必要はあまりないでしょう。

❖ コンパイラドライバオプションとまったく同じ働きをするもの

- -a
- -fcaller-saves
- -fcombine-regs
- -fforce-addr
- -fforce-mem
- -finline-functions
- -fkeep-inline-functions
- -fno-defer-pop
- -fno-function-cse
- -fno-peep-hole
- -fomit-frame-pointer
- -fpcc-struct-return
- -fstrength-reduce
- -funsigned-char
- -fwritable-strings
- -f **レジスタ割り付けオプション**
- -g
- -mshort
- -m68881
- -O
- -o
- -p
- -regparm
- -version
- -W
- -W <文字列>
- -w

❖ gcc.x が間接的に使うもの

- **-hhuman**
  コンパイラを通常の Human68K モードにします。
- **-quiet**
  コンパイラのバーボーズモードを禁止します。

1.2 ……………………………… アセンブラオプション

HAS には，以下のようなオプションがあります。

これらのオプションは，コマンドライン上から直接指定することも，また環境変数 HAS に設定しておくこともできます。また **X680x0 HAS** では，機能拡張にともなっていくつかの新たなオプションスイッチが追加されています[1]。

- "-8"　　**シンボルの識別長の指定**
- "-a"　　**絶対ショートアドレス形式対応モードの指定**[2]
- "-d"　　**全シンボルの外部定義指定**
- "-f"　　**リストファイルのフォーマット指定**
- "-i"　　**インクルードファイルのパス指定**
- "-l"　　**タイトル表示の指定**
- "-m"　　**最大シンボル数の指定**[3]
- "-n"　　**最適化の禁止**
- "-o"　　**オブジェクトファイル名の指定**
- "-p"　　**リストファイルの作成**
- "-q"　　**クイックイミディエイト形式への変換禁止**[4]
- "-r"　　**相対セクション命令の使用許可**[5]
- "-s"　　**シンボルの定義**
- "-t"　　**テンポラリファイルのパス指定**
- "-u"　　**未定義シンボルの外部参照指定**
- "-w"　　**ワーニングレベルの指定**
- "-x"　　**シンボル情報の出力指定**
- "-z"　　**HAS オリジナル機能へのワーニング禁止**[6]

1)追加されたオプションスイッチについては，「X680x0 Develop. & libc II」を参照。

2)X680x0 HAS では，廃止されています。

3)X680x0 HAS では，変更されています。

4)X680x0 HAS では，廃止されています。

5)X680x0 HAS では，廃止されています。

6)X680x0 HAS では，廃止されています。

-8　シンボルの識別長の指定

書式： -8

機能： シンボルの識別長を 8 バイトにします。

解説： シンボルの識別長を 8 バイトまでに制限します。この結果，先頭から 8 バイトが等しい 2 つのシンボルは同じものとして扱われます。
このオプションを指定しないと，シンボルはその長さのすべてを識別します。

例： ソースファイル source.s を，シンボルの識別長を 8 バイトに制限してアセンブルします。

```
A>has -8 source.s
```

次の 3 つのシンボルは，通常すべて異なるものとして扱われますが，-8 オプションをつけてアセンブルすることで，**1.** と **2.** は同じシンボルとして扱われます。

1. symbolABCD
2. symbolAB23
3. symbol0123

-a　絶対ショートアドレス形式対応モードの指定

書式： -a

機能： 絶対ショートアドレス形式のコードを出力します。

解説： ソースファイル中の絶対アドレス参照をしている命令で，アドレスが絶対ショートアドレス形式の範囲に入る場合 ($00000000 ～ $00007FFF または $FFFF8000 ～ $FFFFFFFF) に絶対ショートアドレス形式のコードを出力します。

このオプションを指定しないと，明示的な絶対ショート指定がないかぎり，すべて絶対ロングアドレス形式として扱います。

例： List 1-8 のソースファイル **source.s** を，次のように **-a** オプションをつけてアセンブルします。

```
A>has -a source.s
```

1 行目，2 行目ともに絶対ショート形式の範囲内なので，両方とも絶対ショート形式でアセンブルされます (List 1-9)。

同じソースファイルを **-a** オプションをつけずにアセンブルすると，List 1-10 のように，明示的な絶対ショート形式指定のない 2 行目は絶対ロング形式でアセンブルされます。

List 1-8 ● source.s

```
1:          pea.l   0.w
2:          pea.l   1
```

List 1-9 ● "-a" オプションを指定した場合

```
1:  00000000 48780000                pea.l   0.w
2:  00000004 48780001                pea.l   1
```

List 1-10 ● "-a" オプションを指定しない場合

```
1:  00000000 48780000                pea.l   0.w
2:  00000004 487900000001            pea.l   1
```

変更： **X680x0 HAS** では，つねに絶対ショートアドレス形式対応モードでアセンブルが行われます。そのため，このオプションスイッチは機能しません。

-d　　全シンボルの外部定義指定

書式：　-d

機能：　すべてのシンボルを外部定義します。

解説：　ソースファイル中で定義されているシンボルを，すべて外部定義されているものとして扱います。これによって，そのソースファイル中のすべてのシンボルは，リンクの際に他のモジュールから参照することができるようになります。

このオプションを指定しないと，シンボルは .xdef 疑似命令によって明示的に外部定義宣言をしないかぎり，そのソースファイル内でローカルなものとなります。

例：　ソースファイル source.s を，すべてのシンボルを外部定義してアセンブルします。

```
A>has -d source.s
```

-f　リストファイルのフォーマット指定

書式： -f [<ページングモード>, <マクロ展開モード>,
<表示桁数>,<ページ長>]

機能： リストファイルの出力フォーマットを指定します。

解説： 4つのパラメータによって，リストファイルの出力フォーマットを指定します。すべてのパラメータを省略すると，-f0(ページング禁止)と同じ意味になります。

パラメータの意味は次のとおりです。

- **<ページングモード>**
  リストファイルをページ長ごとに改ページするかどうかを指定する。0 でページングを禁止し，1 でページングを行う。デフォルトは 1 (ページング許可)
- **<マクロ展開モード>**
  ソースファイル中でマクロ展開が行われた場合に，展開された内容をリスト出力するかどうかを指定する。.sall ／.lall 疑似命令と同様の効果をもつ。0 でマクロ展開はリスト中に出力せず (.sall と同等)，1 でリスト中に出力する (.lall と同等)。デフォルトは 0 (.sall と同等)
- **<表示桁数>**
  リストファイルの表示桁数を指定する (.width 疑似命令と同等)。80 ～ 248 の 8 の倍数を指定することができる。デフォルトは 136 桁
- **<ページ長>**
  改ページを行う場合のリストファイルのページ長を指定する (.page 疑似命令と同等)。10 ～ 255 までの値を指定することができる。デフォルトは 56 行

例： ソースファイル source.s をアセンブルして，リストファイル source.prn を作成します。リストファイルはページングを行わず，マクロ展開をリスト中に出力して作成されます。

```
A>has -p -f0,1 source.s
```

-i　インクルードファイルのパス指定

書式：　-i <パス名>

機能：　インクルードファイルを検索するパス名を指定します。

解説：　アセンブル時にインクルードファイルを検索するための **<パス名>** を指定します。

デフォルトでは，インクルードファイルはまずカレントドライブのカレントディレクトリから検索され，そこにファイルが存在しなければ環境変数 **include** に設定されているパスから検索されます。しかし“**-i**”オプションを指定すると，これらよりも先に，まずオプションによって指定されたパス名から検索されます。

これらすべてのパスを検索しても，インクルードファイルが見つからなかったときには，エラー“**file not found error**”が発生します。

例：　インクルードファイルを，最初にディレクトリ **\header** の中から検索するようにアセンブルします。

```
A>has -i\header source.s
```

-l　タイトル表示の指定

書式：　-l

機能：　タイトルの表示を指定します。

解説：　アセンブルを実行するときにタイトル表示を行うようにします。
"-l" オプションが指定されてなく，かつアセンブルがエラーなしで正常に終了したときには，画面に何も表示しません。

例：　ソースファイル **source.s** を，タイトル表示してアセンブルします。

```
A>has -l source.s
X68k High-speed Assembler v2.** Copyright 1990,91,92
No fatal error(s)
```

-m　最大シンボル数の指定

書式：　-m <最大シンボル数>

機能：　使用できる最大のシンボル数を指定します。

解説：　ソースファイル中で使用できるシンボル数の最大値を指定します。
<最大シンボル数> には，1001 ～ 32767 が指定できます。
"-m" オプションの指定がない場合は，最大シンボル数は 10000 個になります。

例：　ソースファイル source.s を，最大シンボル数を 20000 個に指定してアセンブルします。

```
A>has -m20000 source.s
```

変更：　**X680x0 HAS** では，シンボル数の制限はなくなっています。そのため，このオプションスイッチは機能しません。かわりに現在では，アセンブル対象命令セットの指定機能が追加されています。

-n　最適化の禁止

書式： -n

機能： 前方参照の最適化を禁止します。

解説： BRA，Bcc，BSR の各命令のジャンプ先が 8 ビットオフセットの範囲内で届く場合，通常は 16 ビットオフセットを 8 ビットオフセットに変更する処理を行います。しかし "-n" オプションを指定すると，プログラムの前方 (アドレスの増加する方向) へ向かってのジャンプに関しては，この変更を行いません。

このオプションを指定すると，作成されるオブジェクトファイルのサイズは少し大きくなりますが，アセンブルにかかる時間を短縮することができます。

例： List 1-11 のソースファイル source.s を，次のように "-n" オプションをつけてアセンブルします。

```
A>has -n source.s
```

BRA，BSR 両命令ともジャンプ先は 8 ビットオフセットの範囲内ですが，2 行目の BSR 命令は前方参照なので 16 ビットオフセットとなります (List 1-12)。

同じソースファイルを "-n" オプションをつけずにアセンブルすると，List 1-13 のように BSR 命令も 8 ビットオフセットになります。この場合，"-n" オプションをつけた場合に比べて，2 バイト (1 ワード) 小さなオブジェクトファイルが作成されます。

List 1-11● source.s

```
1:  label1:
2:          bsr     label2
3:          bra     label1
4:  label2:
```

List 1-12● "-n" オプションを指定した場合

```
1:  00000000                       label1:
2:  00000000 61000004_00000006             bsr     label2
3:  00000004 60FA_00000000                 bra     label1
4:  00000006                       label2:
```

List 1-13● "-n" オプションを指定しない場合

```
1:  00000000                       label1:
2:  00000000 6102_00000004                 bsr     label2
3:  00000002 60FC_00000000                 bra     label1
4:  00000004                       label2:
```

-o　オブジェクトファイル名の指定

書式：　-o <ファイル名>

機能：　オブジェクトファイルのファイル名を指定します。

解説：　アセンブルによって作成されるオブジェクトファイルのファイル名を指定します。

デフォルトでは，オブジェクトファイルはソースファイル名の拡張子を“.o”に変更した名前で作成しますが，“-o”オプションの使用によって <ファイル名> で指定したオブジェクトファイル名に変更することができます。

例：　ソースファイル source.s をアセンブルします。オブジェクトファイルは，source.o に作成されます。

```
A>has source.s
```

次の例では，オブジェクトファイル名は object.o になります。

```
A>has -oobject.o source.s
```

-p　リストファイルの作成

書式： -p [<ファイル名>]

機能： リストファイルを作成します。

解説： 通常，アセンブルによって作成されるファイルはオブジェクトファイルのみですが，“-p” オプションによってリストファイルの作成を指定できます。

“-p” オプションに <ファイル名> を指定すると，リストファイルはそのファイル名で作成され，省略するとソースファイル名の拡張子を “.prn” に変更した名前で作成されます。

例： ソースファイル source.s をアセンブルして，リストファイル source.prn を作成します。

```
A>has -p source.s
```

次の例では，リストファイルのファイル名は listfile になります。

```
A>has -plistfile source.s
```

-q　クイックイミディエイト形式への変換禁止

書式： -q

機能： クイックイミディエイト形式への変換を禁止します。

解説： 68000 CPU には MOVE, ADD, SUB の各命令に対してイミディエイト値の範囲を制限する代わりに，高速に命令を実行できる MOVEQ, ADDQ, SUBQ 命令（クイックイミディエイト形式）が存在します。

HAS は通常，命令がクイックイミディエイト形式に置き換えることができる場合にその変換を自動的に行いますが，“-q” オプションを指定することでこの変換を禁止します。

例： ソースファイル source.s を，クイックイミディエイト形式への変換を禁止してアセンブルします。

```
A>has -q source.s
```

source.s が List 1-14 のような内容であった場合，“-q” オプションをつけずにアセンブルすると，クイックイミディエイト形式への変換が行われ，実際には MOVE 命令の代わりに MOVEQ 命令のコードが出力されます。“-q” オプションをつけると，ソースファイルどおりに MOVE 命令のコードが出力されます。

List 1-14● source.s

```
1:        move.l  #0,d0
```

変更： **X680x0 HAS** では，クイックイミディエイト形式への変換は，イミディエイト値にサイズ指定を行うことによって禁止します。そのため，このオプションスイッチは機能しません。

-r　相対セクション命令の使用許可

書式： -r

機能： 相対セクション疑似命令の使用を許可します。

解説： SX-Window のプログラム開発を支援するための相対セクション疑似命令 (.rdata, .rbss, .rstack, .rldata, .rlbss, .rlstack, .rcomm, .rlcomm) が使用できるようになります。このオプションをつけずにこれらの疑似命令を使用すると，bad opecode error になります。

例： 相対セクション疑似命令を使用したソースファイル source.has をアセンブルします。

```
A>has -r source.has
```

変更 **X680x0 HAS** では，つねに相対セクション命令を使用できます。そのため，このオプションスイッチは機能しません。

-s　シンボルの定義

書式：　-s <シンボル名> [=<定数>]

機能：　シンボルを定義します。

解説：　シンボルを定義します。<定数> を指定することで，定義するシンボルの値を指定することができます (指定を省略すると，シンボルは定数 0 をもつものとして定義されます)。

例：　ソースファイル source.s を，シンボル "symbol" を定義してアセンブルします[7]。

```
A>has -ssymbol source.s
```

次の例では，同じくシンボル "symbol" を定義してアセンブルしますが，値は 10 進数で 1234 をもちます。

```
A>has -ssymbol=1234 source.s
```

7) "symbol" の値は 0 になります。

-t　テンポラリファイルのパス指定

書式：　-t <パス名>

機能：　テンポラリファイルを作成するパス名を指定します。

解説：　**HAS** ではアセンブルの途中経過で必要となる一時的な情報はすべてメモリ上に記録しますが，メモリ容量が不足している場合にテンポラリファイルを作成することがあります。“-t” オプションでは，このときに実際にテンポラリファイルを作成するパス名を指定します。
“-t” オプションの指定がない場合には，テンポラリファイルは環境変数 temp が設定されていればそのパスに，設定されていないときにはカレントドライブのカレントディレクトリに作成されます。

例：　テンポラリファイルを A:\tmp の中に作成することを指定して，ソースファイル source.s をアセンブルします。

```
A>has -tA:\tmp source.s
```

-u　未定義シンボルの外部参照指定

書式： -u

機能： 未定義のシンボルをすべて外部参照とします。

解説： ソースファイルで定義されていないシンボルが参照された場合，通常は .xref 疑似命令によって明示的に外部参照宣言しないとエラーになります。しかし“-u”オプションを指定すると，この外部参照宣言の有無にかかわらず，未定義のシンボルをすべて自動的に外部参照宣言されているものとして扱います。

例： シンボル“symbol”が定義されずに参照されているソースファイル source.s を次のようにアセンブルすると，シンボルの未定義としてエラーメッセージが表示されます。

```
A>has source.s
source.s            20:   undefined symbol error
undefined symbol(s)
symbol
1 Fatal error(s)
```

次のように“-u”オプションをつけてアセンブルすると，未定義のシンボル“symbol”は外部参照宣言されているものと解釈され，エラーは発生しなくなります。

```
A>has -u source.s
```

-w　　ワーニングレベルの指定

書式：　-w [<ワーニングレベル>]

機能：　ワーニングレベルを指定します。

解説：　アセンブルの際，オブジェクトファイルを作成できないほど致命的ではないエラーに対してはワーニングを出力しますが，“-w” オプションによってワーニングの出力を抑制することができます。

ワーニングは，種類によって 1 ～ 4 の 4 つのレベルに分かれています。デフォルトではすべてのワーニングを出力しますが，<ワーニングレベル> を指定することで，その指定したレベルの値よりも大きいレベルのワーニングの出力を抑制します。

すべてのワーニングを抑制するには，“-w0” と指定します。<ワーニングレベル> の指定を省略すると，“-w2” と同じ意味 (レベル 3, 4 のみ抑制) になります。

各ワーニングレベルの概略は以下のようになります[8)]。

8) ワーニングの内容の詳細については，第 2.2.3 節 (P.174) を参照してください。

- **レベル 1**
  ソースファイルの記述に致命的でない誤りがあるもの，および偶数境界違反に関するもの
- **レベル 2**
  HAS オリジナルの機能に関するもの
- **レベル 3**
  ショートアドレッシング (アドレスレジスタをロングワード以外の単位で扱った場合) に関するもの
- **レベル 4**
  絶対アドレッシングに関するもの

例：　List 1-15 のソースファイル source.s をアセンブルします。

ワーニングを特に抑制しないと，次のようにワーニングメッセージが表示されます。

```
A>has source.s
source.s              1: Warning:  absolute addressing
        move.l  0,d0
source.s              2: Warning:  short addressing
        movea.w d0,a1
```

List 1-15 ● source.s

```
        move.l  0,d0
        movea.w d0,a1
```

このうち 1 行目は絶対アドレス参照によるレベル 4 ワーニング，2 行目はショートアドレッシングによるレベル 3 ワーニングです。次のように -w3 オプションによってレベル 4 ワーニングの出力を抑制すると，表示されるワーニングは 2 行目のもののみになります。

```
A>has -w3 source.s
source.s              2: Warning:  short addressing
        movea.w d0,a1
```

-x　シンボル情報の出力指定

書式： -x [<ファイル名>]

機能： アセンブル時に使用したシンボルの情報を出力します。

解説： アセンブル時に使用したシンボルの情報を出力することを指定します。<ファイル名> が指定されたときはそのファイルに，省略されたときは画面にシンボル情報を出力します。

例： ソースファイル **source.s** をアセンブルして，シンボルの情報を source.sym に出力します。

```
A>has -x source.sym source.s
```

-z　HAS オリジナル機能へのワーニング禁止

書式：　-z

機能：　HAS のオリジナル機能へのワーニングを禁止します。

解説：　拡張子が“.has”でないソースファイル中で HAS のオリジナル機能を使用しても，ワーニングを出力しないようにします。“-w”オプションでワーニング出力レベルを 1 以下にしてもこのワーニングは出力されなくなりますが，“-z”オプションでは他のワーニングには影響を及ぼしません。

このオプションを使用すると，純正アセンブラでのアセンブルができないようなソースファイルに対するチェックができなくなるので注意が必要です。

例：　List 1-16 のソースファイル source.s は HAS オリジナルの命令である .rept 疑似命令を使用しているため，アセンブルすると次のようなワーニングが表示されます。

```
A>has source.s
source.s            1: Warning:  HAS expanded specifications
        .rept   16
```

しかし，次のように“-z”オプションをつけてアセンブルすると，このワーニングが表示されないようになります。

```
A>has -z source.s
```

また，ソースファイルの拡張子を“.has”にすると，“-z”オプションの有無にかかわらずワーニングが表示されないようになります。

List 1-16● source.s

```
        .rept   16
        move.w  (a0)+,(a1)+
        .endm
```

変更：　X680x0 HAS では，オリジナル機能に対してのワーニングは発生しません。そのため，このオプションスイッチは機能しません。

1.3 ………………………………… リンカオプション

HLK には，以下のようなオプションがあります。

これらのオプションは，コマンドライン上から直接指定することも，また環境変数 SILK に設定しておくこともできます。

- “-a”　**実行ファイルの拡張子の省略時に“.x”をつけない**
- “-d”　**外部定義シンボルの登録**
- “-i”　**インダイレクトファイルの指定**
- “-l”　**ライブラリパスの使用**
- “-m”　**最大シンボル数の指定**
- “-o”　**実行ファイル名の指定**[1]
- “-p”　**マップファイルの作成**
- “-s”　**OBJR 形式の情報の作成**
- “-t”　**タイトル表示の指定**
- “-v”　**バーボーズモードの指定**
- “-w”　**ワーニングメッセージの抑制**
- “-x”　**シンボルテーブルの出力禁止**
- “-z”　**-v オプションを無効にする**

1)従来はバグがありましたが，X680x0 HLK ではバグフィックスしています。

-a　実行ファイルの拡張子の省略時に“.x”をつけない

書式： -a

機能： 実行ファイルの拡張子の省略時に，拡張子“.x”をつけないようにします。また，ファイル属性の第7ビットをたてます。

解説： このオプションは，execd[2]に対応させるためのオプションです。このオプションを使用することにより，ファイル名と属性の設定を省くことができます。

例： 実行ファイル **execfile** が作成されます。もし“-a”オプションが指定されていないと，**execfile.x** が作成されます。

```
A>hlk -a -o execfile execfile.o sub1.o sub2.o
A>dir
 ****                 A:\
     4 ファイル        *K Byte 使用中      ***K Byte 使用可能
  ファイル使用量      *K Byte 使用
execfile          o          68  **-**-**    **:**:**
sub1              o          64  **-**-**    **:**:**
sub2              o          64  **-**-**    **:**:**
execfile                     70  **-**-**    **:**:**
A>del execfile
A>hlk -o execfile execfile.o sub1.o sub2.o
A>dir
 ****                 A:\
     4 ファイル        *K Byte 使用中      ***K Byte 使用可能
  ファイル使用量      *K Byte 使用
execfile          o          68  **-**-**    **:**:**
sub1              o          64  **-**-**    **:**:**
sub2              o          64  **-**-**    **:**:**
execfile          x          70  **-**-**    **:**:**
A>
```

2)沖@沖氏 制作のフリーウェア。ファイル属性の第7ビットをたてておくことで，拡張子がなくても実行できるようにする常駐プログラムです。

-d　外部定義シンボルの登録

書式： -d <シンボル>=<サイズ>

機能： 外部定義シンボルを登録します。

解説： 属性[3]が絶対値で，値が <サイズ> の 外部定義シンボル<シンボル> を登録します。このシンボルの定義元のオブジェクトファイル名は *SYSTEM* です。このファイルは **HLK** の内部で作成されるオブジェクトファイルで，実際に存在するファイルではありません。

3)「属性」を参照 (vol.1 第 4.5.1 節)。

例： game.x にはシンボル _MAX_SPEED を，control.x にはシンボル PARAM1 と PARAM2 を外部定義シンボルとして定義します。

```
A>hlk -d _MAX_SPEED=a00 game.o
A>hlk -d PARAM1=5c -d PARAM2=7c control.o
A>
```

リンクされたオブジェクトファイル中に同名の外部定義されたシンボルが存在した場合は，Duplicate definision エラーになります。

```
A>hlk -d DUPLICATE_SYMBOL=10 error.o
Duplicate definition : DUPLICATE_SYMBOL
  in *SYSTEM* error.o
A>
```

-i　インダイレクトファイルの指定

書式： -i <ファイル名>

機能： インダイレクトファイルの指定をします。

解説： **HLK** はオプションやファイル名をコマンドラインから取得していますが，このファイルを指定することにより，コマンドラインの代わりに指定されたファイルの内容を使用するようにします。改行は空白文字とみなします。インダイレクトファイルの中からインダイレクトファイルを指定したり，インダイレクトファイルを複数指定することはできません。

インダイレクトファイルを指定することにより，コマンドラインで指定したファイルやオプションが無効になることはありません。

例： `indirect` ファイルの内容が List 1-17 の場合，`hlk -i indirect` は，

```
hlk main.o sub1.o sub2.o mylib.asyslib.l
```

と同じ意味になります。

インダイレクトファイルの内容は，コマンドラインの後に追加される形になるので注意してください。たとえば，`hlk -i indirect -p` は，

```
hlk -p main.o sub1.o sub2.o mylib.a syslib.l
```

のようになります。この例では，マップファイル名に `main.o` が指定されたことになってしまいます。このまま正常にリンクしてしまうと，`main.o` にマップファイルの内容が書き込まれてしまい，`main.o` の内容は失われてしまいます。

List 1-17 ● indirect

```
main.o sub1.o sub2.o
mylib.a syslib.l
```

-l　　ライブラリパスの使用

書式： -l

機能： オブジェクトファイルやライブラリファイルの検索に，環境変数 lib を使用します。

解説： 指定されたファイルが見つからない場合，環境変数 lib で指定されたディレクトリを捜します。このオプションを指定すると，指定されたファイル，環境変数 lib にファイル名を追加したファイルの順番に検索します。もし，環境変数 lib が設定されていない場合は，Undefined environment variable 'lib' とメッセージを出力して，このオプションを無視します。

例： 環境変数 lib の内容が，a:\lib(または a:\lib\) の場合や指定したファイルが見つからなかった場合に検索されるファイル名の例を Table 1-1 に示します。

Table 1-1● 指定したファイルが見つからなかった場合のファイル名

ファイル名	実際に検索されるファイル名
..\foo.a	a:\lib\..\foo.a
my\bar.o	a:\lib\my\bar.o
a:\foobar.o	a:\lib\a:\foobar.o　(エラー)

-m　最大シンボル数の指定

書式： -m <サイズ>

機能： **HLK** で扱えるシンボルの数を指定します。

解説： 実際にはこのオプションを指定しなくても，メモリが許すかぎりシンボルを扱えるようになっているので，今では指定してもあまり意味がありません。LK との互換性のために用意してあります。
<サイズ> には，202 以上 65536 未満を指定します。

-o　実行ファイル名の指定

書式： -o <ファイル名>

機能： 作成される実行ファイル名を指定します。

解説： 実行ファイル名は，最初にリンクされたオブジェクトファイルの拡張子を“.x”に変更した名前になりますが，このオプションを指定することにより変えることができます。拡張子が指定されていない場合は，“.x”が追加されます[4]。

4)ただし，-a オプションが指定されていない場合にかぎります。

例： 実行ファイル **exec.x** を作成します。

```
A>hlk -o exec main.o
A>
```

実行ファイル **main.x** を作成します。

```
A>hlk main.o
A>
```

実行ファイル **exec.xx** を作成します。

```
A>hlk -o exec.xx main.o
A>
```

追 記： 従来はバグがありましたが，**X680x0 HLK** ではバグフィックスしています。

-p　　マップファイルの作成

書式：　-p <マップファイル名>

機能：　マップファイルを作成します。

解説：　指定されたファイル名のマップファイル[5]を作成します。<マップファイル名> を省略した場合は，実行ファイル名の拡張子を “.map” に置き換えたものになります。

5)「マップファイル」を参照（Vol.1 第 4.1.1 節）。

例：　マップファイル **main.map** を作成します。

```
A>hlk main.o -p
A>
```

マップファイル **mapfile.map** を作成します。

```
A>hlk owner.o -p mapfile
A>
```

-s　OBJR 形式の情報の作成

書式： -s

機能： GCC で作成した SX–Window の OBJR 形式の実行ファイルに必要な情報を作成します。

解説： 外部定義シンボル ___rsize が定義され，セクション情報が実行ファイルに埋め込まれます。___rsize には，rdata ～ rstack セクションの大きさが代入されます。属性は絶対値になります。セクション情報の内容は Fig.1-1 のようになっています。

GCC で作成した SX–Window の OBJR 形式の実行ファイルを作成する時は，必ず指定してください。この OBJR 形式の実行ファイルは，外部定義シンボル ___rsize とセクション情報を参照して，リロケータブルセクションの初期化を行います。もちろん，この初期化を行うルーチンがリンクされていなければ，プログラムは正常に動きません。

例： OBJR 形式の実行ファイルに必要なセクション情報が入っている window.x が作成されます。

```
A>gcc -SX -c window.c event.c
A>hlk -s window.o event.o a:\lib\_sxlib.a a:\lib\sxlib.a
A>
```

+00 +01 +02 +03	+04 +05 +06 +07	+08 +09 +0a +0b	+0c +0d +0e +0f
text size	data size	bss size	common size
stack size	rdata size	rbss size	rcommon size
rstack size	rldata size	rlbss size	rlcommon size
rlstack size	roffset table size	reserved	

Fig. 1-1● セクション情報のフォーマット

-t　タイトル表示の指定

書式：　-t

機能：　起動時にタイトルを表示します。

解説：　起動時のタイトルを表示します。このオプションが指定されていない場合は，タイトルを表示しません。LK のように起動時にタイトルを表示することによって，リンク処理を始めたことを確認したいときなどに使用します。

例：　-t オプションを指定した場合は，タイトルが表示されます。

```
A>hlk one.o two.o
A>
A>hlk -t one.o two.o
X68k SILK Hi-Speed Linker v2.26 Copyright 1989-92 SALT
A>
```

-v　　バーボーズモードの指定

書式： -v

機能： **HLK** をバーボーズモードで動作するようにします。

解説： バーボーズモードを指定すると，**HLK** が行っている動作を報告するようになります。このオプションによって，リンカが現在どのような処理を行っているのかを確認することができます。

例： -v オプションを指定している行は，現在どのような処理を行っているのかを報告しています。

```
A>hlk main.o sub1.o sub2.o
A>
A>hlk -v main.o sub1.o sub2.o
Linked : main.o
Linked : sub1.o
Linked : sub2.o
Making executable file...
A>
```

-w　ワーニングメッセージの抑制

書式： -w

機能： ワーニングメッセージを出力しないようにします。

解説： g++ (GNU C++ Compiler) でコンパイルしたオブジェクトファイルをリンクするとワーニングが頻繁に出ますが，このオプションを指定することでワーニングメッセージを出力しないようにできます。

例： -w オプションを指定している場合は，ワーニングメッセージが出力されません。

```
A>hlk warning.o cause.o
Warning, duplicate definition : _WARNING
A>
A>hlk -w warning.o cause.o
A>
```

-x　シンボルテーブルの出力禁止

書式：　-x

機能：　シンボルテーブル，シンボリックデバッグ情報を作成しないようにします。

解説：　このオプションを指定しない場合は，シンボルテーブルやシンボリックデバッグ情報が作成されます。デバッグする必要がなくなった場合，これらの情報は不要になります。このオプションを使用して再リンクすることにより，不要になった情報を作成しないことで実行ファイルの大きさを小さくすることができます[6]。

例：　**-x** オプションを指定した実行ファイルはシンボルテーブル，シンボリックデバッグ情報がないため，ファイルのサイズが小さくなっています。

```
A>hlk symbols.o
A>dir symbols.x
 ****                    A:\
     1 ファイル       **K Byte 使用中      ***K Byte 使用可能
  ファイル使用量     **K Byte 使用
symbols             x          418  **-**-**  **:**:**
A>
A>hlk -x symbols.o
A>dir symbols.x
 ****                    A:\
     1 ファイル       **K Byte 使用中      ***K Byte 使用可能
  ファイル使用量     **K Byte 使用
symbols             x          338  **-**-**  **:**:**
A>
```

6)xpack.x や strip.x などのフリーウェアのように，再リンクしなくても実行ファイルのシンボルテーブルやシンボリックデバッグ情報を削除するツールがあります。

-z　　-v オプションを無効にする

書式： -z

機能： バーボーズモードで動作しないようにします。

解説： このオプションを使用すると，どこに -v オプションが指定されてもバーボーズモードになりません。たとえば，GCC でバーボーズモードにしてコンパイル／リンクを行うときに，リンカの部分はバーボーズモードにしたくない場合，環境変数 SILK または silk にこのオプションを追加しておくとよいでしょう。

例： このオプションを指定するとバーボーズモードが無視されるので，-v オプションを指定しても実行経過が表示されません。

```
A>hlk -v main.o sub.o -o game.x
Linked : main.o
Linked : sub.o
Making executable file...
A>hlk -z -v main.o sub.o -o game.x
A>
```

1.4 ……………………………… デバッガオプション

GDB には，以下のようなオプションがあります。

これらのオプションは，コマンドライン上から直接指定することも，また環境変数 GDB_OPTION に設定しておくこともできます。

- “-batch” **バッチ処理**
- “-cd” **ワーキングディレクトリの指定**
- “-command” **コマンドファイルの指定**
- “-directory” **ソースディレクトリのパス指定**
- “-exec” **デバッグ対象プログラムの指定**[1]
- “-epoch”，“-fullname” **Emacs を使用する**[2]
- “-help” **ヘルプメッセージの表示**
- “-mem” **メモリの設定**
- “-nx” **初期化ファイルを読み込まない**
- “-quiet” **タイトルの非表示**
- “-remote”，“-r” **リモートコンソールモードで起動する**
- “-se” **デバッグ対象プログラムの指定**[3]
- “-symbols” **シンボル情報のファイルの指定**[4]
- “-swap” **スクリーンスワップモードで起動する**
- “-tty” **標準入出力先の指定**

1)X680x0 GDB では，廃止されています。

2)X680x0 GDB では，廃止されています。

3)X680x0 GDB では，廃止されています。

4)X680x0 GDB では，廃止されています。

-batch バッチ処理

書式： -batch

機能： バッチモードで起動します。

解説： “-command” オプション に指定したコマンドファイルのすべての処理が終了した後，exit コード 0 で終了します。コマンドファイル内の **GDB** のコマンド実行中にエラーが発生した場合は，0 以外の exit コードで終了します。

例： コマンドファイル “cmd” に従ってバッチ処理を行います。

```
A>gdb -batch -command=cmd smp.x
Breakpoint 1 at 0x108: file smp.c, line 19.
Breakpoint 2 at 0x1f0: file smp.c, line 20.

Breakpoint 1, main () at smp.c:19
19   shellsort (data, 9);
data[] = {4 3 5 8 9 6 1 7 2 }
$1 = 2

Breakpoint 2, main () at smp.c:20
20 }
data[] = {1 2 3 4 5 6 7 8 9 }
$2 = 2
A>
```

-cd　ワーキングディレクトリの指定

書式： -cd= <パス名>

機能： ワーキングディレクトリを指定します。

解説： 起動時に指定したワーキングディレクトリに切り替えます。

例： 次の画面は，起動後にワーキングディレクトリを“c:\gdb-4.4”に切り替えた例です。

```
A>pwd
A:\cmd
A>gdb -cd=c:\gdb-4.4
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb) pwd
Working directory c:\gdb-4.4
 (canonically C:\gdb-4.4).
(gdb)
```

-command　コマンドファイルの指定

書式：　-command= <ファイル名>

機能：　コマンドファイルを指定します。

解説：　指定したファイルから **GDB** のコマンドを読み込み実行します。

例：　起動後に指定したコマンドファイル “test” を実行してみました。

```
A>gdb -command=test
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
Breakpoint 1 at 0x108: file smp.c, line 19.
(gdb)
```

-directory　ソースディレクトリのパス指定

書式：　-directory= <ディレクトリ名>

機能：　ソースディレクトリを指定します。

解説：　ソースファイルがおいてあるディレクトリを指定します。

例：　ソースサーチパスを "c:\src\orig" に設定し，起動しました。

```
A>gdb -directory=c:\src\orig
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb) show dir
Source directories searched: B:\src\gnu;$cdir;$cwd
(gdb)
```

-exec　デバッグ対象プログラムの指定

書式：　-exec= <ファイル名>

機能：　デバッグ対象プログラム名を指定します。

解説：　実行ファイルとシンボル情報ファイルが分かれているときに使います。

例：　デバッグ対象ファイルを “test.x” に設定し，起動します。

```
A>gdb -exec=test.x
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

GDB が起動した後にデバッグ対象プログラムを指定するには，次のようにします。

```
(gdb) exec-file test.x
(gdb)
```

変更：　.x 形式の実行ファイルは，シンボリックデバッグ情報を含むことができます。そのため，このオプションスイッチは意味がないので廃止しました。

-epoch／-fullname　Emacs を使用する

書式： -epoch
-fullname

機能： Emacs を **GDB** のフロントエンドに使うときに指定します。

解説： **X68000 GDB** では意味がありません。

変更： **Human68k** がマルチタスクになることを期待して残しておいたオプションスイッチですが，今のところ意味がないので，**X680x0 GDB** では廃止しました。

-help　ヘルプメッセージの表示

書式：　-help

機能：　コマンドラインオプションのヘルプメッセージを表示します。

解説：　コマンドラインオプションをすべて説明つきで表示します。

例：　次の画面は，コマンドラインオプションを表示させたものです。

```
A>gdb -help
This is GDB, the GNU debugger.  Use the command
    gdb [options] [executable [core-file]]
to enter the debugger.

Options available are:
  -help              Print this message.
  -quiet             Do not print version number on sta
rtup.
  -fullname          Output information used by emacs-G
DB interface.
  -epoch             Output information used by epoch e
macs-GDB interface.
  -batch             Exit after processing options.
  -nx                Do not read .gdbinit file.
  -tty=TTY           Use TTY for input/output by the pr
ogram being debugged.
  -cd=DIR            Change current directory to DIR.
  -directory=DIR     Search for source files in DIR.
  -command=FILE      Execute GDB commands from FILE.
  -symbols=SYMFILE   Read symbols from SYMFILE.
  -exec=EXECFILE     Use EXECFILE as the executable.
  -se=FILE           Use FILE as symbol file and execut
able file.
  -core=COREFILE     Analyze the core dump COREFILE.
  -b BAUDRATE        Set serial port baud rate used for
 remote debugging
  -heap=SIZE         ＧＤＢプロセスが使用するメモリーの設定
  -remote            リモートコンソールモードで起動
  -swap              デバッガとアプリケーションの画面を切り替え
るモード

For more information, type "help" from within GDB, or
consult the GDB manual (available as on-line info or a
printed manual).
A>
```

-mem　メモリの設定

書式：　-mem= <サイズ>

機能：　GDB プロセスが使用するメモリを設定します。

解説：　<サイズ> はキロバイト単位で指定します。

解説：　このオプションは，デバッグ中に GDB のプロセスが「メモリが足りなくなった」ことを示すメッセージを表示した場合に指定します。GDB プロセスのためにあらかじめ確保する領域は，メモリのフリーサイズに依存します。

例：　メモリサイズを 512K バイトに設定してみましょう。

```
A>gdb -mem=512
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

-nx　初期化ファイルを読み込まない

書式：　**-nx**

機能：　初期化ファイルを読み込まないで起動します。

解説：　**GDB** が起動時に読み込む初期化ファイルは，“**.gdbinit**” です。

例：　次の場面は，初期化ファイルを読み込まないで起動した例です。

```
A>gdb -nx
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

-quiet　タイトルの非表示

書式：　-quiet

機能：　起動メッセージを表示しません。

解説：　GDBのタイトル，バージョンナンバー等を表示しないで起動します。

例：　起動メッセージを表示しないで，起動した例です。

```
A>gdb -quiet
(gdb)
```

-remote／-r リモートコンソールモードで起動する

書式： -remote

-r

機能： RS–232C を利用したリモートコンソールモードで起動します。

解説： **X68000** に RS–232C ポートで通信の行えるコンピュータ[5]を接続し，接続したターミナルのコンソールを利用してデバッガを操作することで，**X68000** の画面を乱すことなくデバッグ作業を行うことができるモードです。このモードを使用するには，RS–232C ポートどうしを接続するためのクロスケーブルと，コンピュータがもう 1 台必要です。リモートコンソールモードでデバッグするには，起動時にオプション“**-remote**”または“**-r**”を指定します[6]。

この後，**GDB** の操作は接続したターミナルから行うことになります。

例： リモートコンソールモードで起動した場合，次のようになります。起動メッセージはターミナル側に表示されます。

```
A>gdb -remote
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

5)ターミナルといいます。

6)起動する前に必ず，両方のコンピュータの通信速度を合わせておいてください。

-se　デバッグ対象プログラムの指定

書式：　-se= <ファイル名>

機能：　デバッグ対象プログラム名を指定します。

解説：　シンボルファイルが実行ファイルに含まれている場合，このコマンドで指定します。

例：　“test.x” がデバッグ対象プログラムならば，次のような起動画面になります。

```
A>gdb -se=test.x
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

起動後にコマンドで指定するには次のようにします。

```
(gdb) file test.x
(gdb)
```

変更：　GDB では，デバッグ対象プログラムのファイル名をその他のオプションスイッチの後ろに単独で指定することができます。そのため，このオプションスイッチは廃止しました。

-symbols シンボル情報ファイルの指定

書式： -symbols= <ファイル名>

機能： シンボル情報を読み込むファイルを指定します。

解説： 実行ファイルとシンボル情報ファイルが分かれているときに使用します。

例： シンボルファイル "test.x" を指定して起動しました。

```
A>gdb -symbols=test.x
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

起動後にコマンドで指定するには，次のようにします。

```
(gdb) symbol-file test.x
(gdb)
```

変更： -exec スイッチ同様，.x ファイルがシンボリックデバッグ情報を含みます。そのため，このオプションスイッチは意味がないので廃止しました。

-swap　スクリーンスワップモードで起動する

書式：　-swap

機能：　スクリーンスワップモードで起動します。

解説：　**X68000** の画面を乱さないために，**GDB** の画面とデバッグ対象プログラムの画面とを切り替えて表示するモードです。ただし，それぞれの画面のイメージを退避するため，メインメモリを消費します[7]。

このモードは，RS–232C ポートに接続するコンピュータがなくても使用することができます。

スクリーンスワップモードでデバッグを行うには，起動時に“**-swap**”を指定して起動します。

7)約 132K バイトです。

例：　スクリーンスワップモードで起動した例です。

```
A>gdb -swap
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

-tty　標準入出力先の指定

書式：　-tty= <デバイス名>

機能：　デバッグ対象プログラムの標準入出力先を指定します。

解説：　デバッグ対象プログラムの標準入出力を“aux”から行いたい場合などに指定します。

例：　デバッグ対象プログラムの標準入出力を，“aux”に設定して起動します。

```
A>gdb -tty=aux
GDB is free software and you are welcome to distribute
 copies of it under certain conditions; type "show cop
ying" to see the conditions.There is absolutely no war
ranty for GDB; type "show warranty" for details.GDB 4.
* X6_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

Chapter 2
診断メッセージ

本章では，各ツールから出力するエラーメッセージとワーニングメッセージについて解説します。

2.1 …………………………… GCC 診断メッセージ

このセクションでは，GCC のエラーおよびワーニングメッセージについて説明します。参考として，まったく同一ソースを XC Ver. 2.10 でコンパイルした場合のエラーも同時に掲載しておきました。

2.1.1 宣言に関するエラー

ここでは，宣言関係のエラーメッセージを集めました。宣言関係のエラーは，C 言語に慣れていない間は頻発するエラーです。

診断： '*Ident*' が別の宣言をされました
'*Ident*' が再宣言されました
'*Ident*' が別記憶クラス宣言されました

解説： `int` と宣言した *Ident* を `double` と再度宣言したり，関数と宣言した *Ident* を変数に再宣言したりすると発生します。首尾一貫して同じ宣言をしてください。

List 2-1 ● エラー例 (err001.c)

```
1:  int foo;
2:  int foo ();
```

```
err001.c:       2: Error  : 'foo' が別の宣言をされました
err001.c:       1: Error  : 'foo' の前宣言位置です
```

XC Ver. 2.10

```
err001.c      2 :Error       29:identifier redeclaration.
```

診断： ‘*Ident*’ が built-in 関数と衝突しました

解説： GCC の拡張である内部関数 __builtin_alloca () 等を内部宣言と異なる宣言をすると発生します。

List 2-2 ● エラー例 (err002.c)

```
extern __builtin_alloca (void);
```

```
err002.c:       1: Error   : ‘__builtin_alloca’ が built-in 関数と衝突しました
```

診断： ‘*Ident*’ と衝突しています

解説： たとえば void と宣言した関数を，int として定義しようとした場合に発生します。たいていこのエラーメッセージの後に，前の宣言位置がコンパイラによって示されます。ヘッダでの宣言と実体の定義が矛盾するとき多発します。

List 2-3 ● エラー例 (err003.c)

```
void foo (int);
int
foo (void)
{
}
```

```
err003.c: In function foo:
err003.c:       4: Error   : ‘foo’ と衝突しています
err003.c:       1: Error   : ‘foo’ の前宣言位置です
```

XC Ver. 2.10

```
err003.c      4 :Error       29:identifier redeclaration.
```

診断： 省略引数は空の引数宣言とはマッチしません

解説： int foo (); という宣言と int foo (void) で始まる関数定義とは **ANSI C** では別宣言です。前者は“伝統的な宣言”で，後者は **ANSI C** のプロトタイプです。

List 2-4 ● エラー例 (err004.c)

```
int foo ();
int
foo (int,...)
{
}
```

```
err004.c: In function foo:
err004.c:        5: Error  : 'foo' と衝突しています
err004.c:        5: Error  : 省略引数は空の引数宣言とはマッチしません
err004.c:        1: Error  : 'foo' の前宣言位置です
```

特記事項

XC Ver. 2.10 ではエラーになりません。

診断： '*Ident*' の前宣言位置です

解説： 宣言関係のエラーが発生した場合に，現在の宣言と異なる宣言を行った位置を示すエラーです。

診断： '*Ident*' が宣言の前に使われました

解説： 筆者はこのエラーが発生したのを見たことがありません。発生する場合があるらしいのですが・・・。

診断： ラベル '*Ident*' が複数あります

解説： goto で用いられるラベルに同じ識別子をもつラベルが存在すると発生します。goto 文での参照の有無には無関係です。ラベルは同一関数内で重複できません。

List 2-5 ● エラー例 (err005.c)

```
void foo (void)
{
Label0:
          ;
Label0:
          ;
}
```

```
err005.c: In function foo:
err005.c:       3: Error   : ラベル'Label0' が複数あります
```

XC Ver. 2.10

エラーメッセージの後で画面にゴミを出します。

```
line    8 redifinittion error
```

診断： label *Label* がありません

解説： goto 文で存在しない *Label* を参照すると発生します。あるいは別の関数で定義されている *Label* を参照している場合に発生します。ラベルのタイプミスの可能性が大です。

List 2-6 ● エラー例 (err006.c)

```
int foo ()
{
  goto Label;
LabeL:
  ;
}
```

```
err006.c: In function foo:
err006.c:       6: Error   : label Label がありません
```

XC Ver. 2.10

```
err006.c      6 :Error      42:undefined identifier.
err006.c      6 :Warning    27:label is never used in function.
err006.c      6 :Error      42:undefined identifier.
```

診断： '*Ident*' は違法なタグです

解説： struct として宣言したタグを新たに union として宣言すると発生します。一度宣言した構造体タグを，別の種類の構造体タグとして使うと発生します。タグ名称を変えてください。

List 2-7● エラー例 (err007.c)

```
struct foo
{
  int x;
  int y;
} bufst;
union foo
{
  int x;
  char *y;
} bufuni;
```

```
err007.c:       7: Error  : 'foo' は違法なタグです
```

XC Ver. 2.10

```
err007.c     7 :Error   30:struct/union/enum tag name error : foo.
```

診断：　'*Ident*' のサイズは不明です

解説：　不完全な構造体の変数を定義すると発生します。

List 2-8● エラー例 (err008.c)

```
struct foo;
struct foo bar;
```

```
err008.c:       2: Error  : 'bar' のサイズは不明です
```

XC Ver. 2.10

```
err008.c     1 :Error      71:structure declaration error.
err008.c     2 :Error      14:undefined struct/union name.
err008.c     4 :Error      14:undefined struct/union name.
```

診断：　'*Ident*' のサイズが定数ではありません

解説：　関数内部での自動変数でない配列宣言で，サイズが定数でない場合に発生します。

List 2-9 ● エラー例 (err009.c)

```
void foo (int a)
{
    static int buf0[a];
    extern int buf1[a];
}
```

```
err009.c: In function foo:
err009.c:       3: Error   : 'buf0' のサイズが定数ではありません
err009.c:       4: Error   : 'buf1' のサイズが定数ではありません
```

XC Ver. 2.10

```
err009.c     3 :Error       57:constant expression required.
err009.c     3 :Warning      9:data size 1Mbytes over.
err009.c     4 :Error       57:constant expression required.
```

診断： '*Ident*' が複数のデータタイプを持っています

解説： int double foo; のようなまちがった宣言で発生します。

List 2-10 ● エラー例 (err010.c)

```
int double x;
int float  y;
```

```
err010.c:       1: Error   : 'x' が複数のデータタイプを持っています
err010.c:       2: Error   : 'y' が複数のデータタイプを持っています
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： '*Ident*' は typedef か built in type できません

解説： typedef しようとするタイプの識別子が，変数のタイプ宣言を表す識別子でない場合に発生します。筆者はこのエラーが発生したのを見たことがありません。

診断：　'*Ident*' の long, short, signed , unsigned は不正です

解説：　signed float x; のような誤った宣言で発生します。float タイプには long, short, signed, unsigned は指定できません。double タイプには short, signed, unsigned は指定できません。

List 2-11 ● エラー例 (err011.c)

```
1:  signed double x;
2:  unsigned float y;
```

```
err011.c:      1: Error  : 'x' の long, short, signed , unsigned は不正です
err011.c:      2: Error  : 'y' の long, short, signed , unsigned は不正です
```

XC Ver. 2.10

```
err011.c      1 :Warning    22:signed type specifier illegal.
err011.c      2 :Warning     7:unsigned type specifier illegal.
```

診断：　'*Ident*' に long と short 双方指定されています

解説：　short long x; のような宣言で発生します。

List 2-12 ● エラー例 (err012.c)

```
1:  short long int x;
```

```
err012.c:      1: Error  : 'x' に long と short 双方指定されています
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断：　char '*Ident*' に long, short 指定されています

解説：　long char x; のような宣言で発生します。

List 2-13● エラー例 (err013.c)

```
short char x;
long char  y;
```

```
err013.c:       1: Error   : char 'x' に long, short 指定されています
err013.c:       2: Error   : char 'y' に long, short 指定されています
```

XC Ver. 2.10

2 行目は黙って通します。

```
err013.c     1 :Warning     6:short type specifier illegal.
```

診断： '*Ident*' に signed と unsigned 双方指定されています

解説： signed unsigned x; のような宣言で発生します。

List 2-14● エラー例 (err014.c)

```
signed unsigned int x;
```

```
err014.c:       1: Error   : 'x' に signed と unsigned 双方指定されています
```

XC Ver. 2.10

```
err014.c     1 :Warning     7:unsigned type specifier illegal.
```

診断： '*Ident*' の記憶クラスが複数です

解説： static extern x; のような宣言で発生します。

List 2-15● エラー例 (err015.c)

```
static extern x;
static auto   y;
```

```
err015.c:        1: Error   : 'x' の記憶クラスが複数です
err015.c:        2: Error   : 'y' の記憶クラスが複数です
```

XC Ver. 2.10

```
err015.c      1 :Error       10:illegal use of storage class : static.
err015.c      2 :Error       10:illegal use of storage class : static.
err015.c      2 :Error        7:storage class illegal : auto.
```

診断： 関数の記憶クラスが 'auto' です
関数の記憶クラスが 'register' です
関数の記憶クラスが 'typedef' です
関数の記憶クラスが 'common' です
関数の記憶クラスが 'remote' です
関数の記憶クラスが 'relocate' です

解説： 関数の記憶クラスに auto, register, typedef, common, remote, relocate を指定すると発生します。

List 2-16 ● エラー例 (err016.c)

```
auto int foo (void)
{}
register int bar (void)
{}
```

```
err016.c:        2: Error   : 関数の記憶クラスが'auto' です
err016.c:        4: Error   : 関数の記憶クラスが'register' です
```

XC Ver. 2.10

```
err016.c      1 :Error        7:storage class illegal : auto.
err016.c      2 :Warning     15:function return value mismatch.
err016.c      3 :Error        7:storage class illegal : register.
err016.c      4 :Warning     15:function return value mismatch.
```

診断： top-level での 'auto' は不正です

解説： 関数の外で変数を auto 宣言すると発生します。

List 2-17● エラー例 (err017.c)

```
1:  auto int x;
```

```
err017.c:       1: Error   : top-level での'auto' は不正です
```

XC Ver. 2.10

```
err017.c     1 :Error       7:storage class illegal : auto.
```

診断： '*Ident*' が void の配列です
'*Ident*' が関数の配列です

解説： 構文上，これらの配列は宣言できますが，意味的にはまちがいです。

List 2-18● エラー例 (err018.c)

```
1:  void foo[10];
2:  int (bar[10])(void);
```

```
err018.c:       1: Error   : 'foo' が void の配列です
err018.c:       2: Error   : 'bar' が関数の配列です
```

XC Ver. 2.10

このエラーメッセージは誤植ではありません。

```
line    2 illegal value error
line    3 illegal value error
```

診断： 配列 '*Ident*' のサイズが整数ではありません

解説： 配列宣言で，配列の大きさが整数でない場合に発生します。

List 2-19● エラー例 (err019.c)

```
1:  int foo[10.0];
```

```
err019.c:        1: Error  : 配列'foo' のサイズが整数ではありません
```

XC Ver. 2.10

```
err019.c      1 :Error      57:constant expression required.
```

診断： 配列 '*Ident*' のサイズが負です

解説： 配列宣言で，配列の大きさを負の定数にした場合に発生します。

List 2-20 ● エラー例 (err020.c)

```
1:  int foo[-10];
```

```
err020.c:        1: Error  : 配列'foo' のサイズが負です
```

XC Ver. 2.10

```
err020.c      1 :Warning    24:size of array is negative.
```

診断： 関数 '*Ident*' const で volatile な関数は違法です

解説： これは GCC 拡張形式で const 属性と同時に volatile 属性を指定した場合に発生します。GCC のオリジナルソースで作成された GCC では，このエラーは発生しません。

List 2-21 ● エラー例 (err021.c)

```
1:  const volatile foo (void);
```

```
err021.c:        1: Error  : 関数'foo' const で volatile な関数は違法です
```

診断： 関数 ‘*Ident*’ 関数を返すことはできません

解説： 関数は関数を返すことはできません。そのような宣言は違法です。

List 2-22● エラー例 (err022.c)

```
void foo (void)();
```

```
err022.c:        1: Error   : 関数‘foo’ 関数を返すことはできません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 関数 ‘*Ident*’ 配列を返すことはできません

解説： 関数は配列を返すことはできません。ただし構造体で，配列をくるむことで疑似的に返すことはできます。

List 2-23● エラー例 (err023.c)

```
int (foo ())[];
```

```
err023.c:        1: Error   : 関数‘foo’ 配列を返すことはできません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 関数 ‘*Ident*’ はプロトタイプ必須です

解説： SXCALL 属性関数，および DOSCALL 属性関数はプロトタイプ宣言が必須です。

List 2-24● エラー例 (err024.c)

```
DOSCALL foo ();
```

```
err024.c:        1: Error   : 関数‘foo’ はプロトタイプ必須です
```

診断： ポインタ宣言が不正です

解説： int *const foo; のような型修飾子をもったポインタ宣言に誤りがあった場合に発生しますが，現在の **GCC** の型修飾子は const, volatile 以外はないので発生しないエラーのはずです。誤った宣言は，このエラーの前に文法違反等のエラーになるはずだからです。

診断： 変数 '*Ident*' が void に宣言されました

解説： void foo; のように変数を void に宣言しています。void である変数はありません。

List 2-25 ● エラー例 (err025.c)

```
void  foo;
```

```
err025.c:       1: Error   : 変数'foo' が void に宣言されました
```

XC Ver. 2.10

```
err025.c      1 :Warning      4:void type declarator.
```

診断： field '*Ident*' が関数に宣言されています

解説： 共用体や構造体のメンバが関数として宣言されています。C++ では，構造体の中に関数を宣言することができます。

List 2-26 ● エラー例 (err026.c)

```
struct foo
{
  int bar (void);
  int x;
} buf;
```

```
err026.c:       3: Error   : field'bar' が関数に宣言されています
```

XC Ver. 2.10

```
err026.c     3 :Error      31:storage class error at function.
```

診断： field‘*Ident*’が不完全です

解説： ネストした構造体のメンバである構造体が不完全な場合に発生します。

List 2-27● エラー例 (err027.c)

```
struct foo;
struct bar
{
  struct foo x;
  int y;
} buf;
```

```
err027.c:      4: Error   : field‘x’が不完全です
```

XC Ver. 2.10

```
err027.c     1 :Error      71:structure declaration error.
err027.c     4 :Error      14:undefined struct/union name.
```

診断： 関数‘*Ident*’の記憶クラスが不正です

解説： 関数に不正な記憶クラス指定が行われているときに発生します。

List 2-28● エラー例 (err028.c)

```
auto int foo (void);
```

```
err028.c:      1: Error   : top-level での‘auto’は不正です
err028.c:      1: Error   : 関数‘foo’の記憶クラスが不正です
```

XC Ver. 2.10

```
err028.c     1 :Error       7:storage class illegal : auto.
```

診断： SXCALL 関数は inline にできません

解説： SXCALL 属性関数を inline に宣言すると発生します。このエラーは **X68000 GCC** 拡張です。

List 2-29 ● エラー例 (err029.c)

```
inline SXCALL int foo (void);
```

```
err029.c:        1: Error   : SXCALL 関数は inline にできません
```

診断： 変数 '*Ident*' を SXCALL クラス指定しています

解説： SXCALL 属性を変数に対して用いると発生します。このエラーは **X68000 GCC** 拡張です。

List 2-30 ● エラー例 (err030.c)

```
SXCALL int foo;
```

```
err030.c:        1: Error   : 変数'foo' を SXCALL クラス指定しています
```

診断： 引数 '*Ident*' が不完全です

解説： 関数の引数に構造体を指定したとき，その構造体が不完全な場合に発生します。構造体のポインタを引数にする場合は，不完全でもエラーにはなりません。

List 2-31 ● エラー例 (err031.c)

```
struct foo;
int bar (struct foo x)
{}
```

```
err031.c:        3: Error   : 引数'x' が不完全です
```

XC Ver. 2.10

```
err031.c     1 :Error      71:structure declaration error.
err031.c     3 :Error      14:undefined struct/union name.
```

診断： 引数の void は1つしか存在できません

解説： int foo (void, void); のような宣言をすると発生します。

List 2-32 ● エラー例 (err032.c)

```
1:  int foo (void, void);
```

```
err032.c:       1: Error   : 引数の void は 1 つしか存在できません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： union *Ident* が再定義されました
struct*Ident* が再定義されました

解説： 構造体／共用体が，同一識別子で異なる定義をした場合に発生します。

List 2-33 ● エラー例 (err033.c)

```
1:  struct foo { int x; };
2:  struct foo { int x; int y;};
```

```
err033.c:       2: Error   : struct foo が再定義されました
```

XC Ver. 2.10

```
err033.c     2 :Error 30:struct/union/enum tag name error : foo.
```

診断： bit-field '*Ident*' の幅が整数ではありません

解説： ビットフィールドの幅に，整数定数以外を用いた場合に発生します。

List 2-34 ● エラー例 (err034.c)

```
struct {
    unsigned bar:10.0;
} buf;
```

```
err034.c:       2: Error   : bit-field 'bar' の幅が整数ではありません
```

XC Ver. 2.10

```
err034.c       2 :Error       57:constant expression required.
err034.c       2 :Error       37:field width overflow.
```

診断： bit-field '*Ident*' に不当な type があります

解説： ビットフィールドに整数タイプ以外が指定されています。**GCC** ではビットフィールドにすべての整数タイプを指定できますが，**ANSI C** では int か unsigned，unsigned int だけです。

List 2-35 ● エラー例 (err035.c)

```
struct {
  double bar:4;
} buf;
```

```
err035.c:       2: Error   : bit-field 'bar' に不当な type があります
```

XC Ver. 2.10

```
err035.c       2 :Error       38:field type error.
```

診断： bit-field '*Ident*' のサイズが 0 です

解説： 幅のないビットフィールドを作成しようとしています。

List 2-36 ● エラー例 (err036.c)

```
struct {
  unsigned bar:0;
} buf;
```

```
err036.c:       2: Error   : bit-field 'bar' のサイズが 0 です
```

XC Ver. 2.10

```
err036.c      3 :Error       14:undefined struct/union name.
err036.c      4 :Error       14:undefined struct/union name.
```

診断： メンバ '*Ident*' が 2 重です

解説： 構造体／共用体でメンバ名が重複しています。同一構造体の中ではすべて異なった名前を使ってください。異なる構造体で同一名称が使用されているのは違反ではありません。

List 2-37 ● エラー例 (err037.c)

```
struct {
  int bar;
  int bar;
} buf;
```

```
err037.c:       3: Error   : メンバ'bar' が 2 重です
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 'enum *Ident*' が再宣言されています

解説： enum を同じ名称で別の宣言を行っています。別の名前で宣言してください。

List 2-38 ● エラー例 (err038.c)

```
enum foo { R, G, B };
enum foo { X, Y, Z };
```

```
err038.c:        2: Error  : 'enum foo' が再宣言されています
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： enum '*Ident*' の値が整数定数ではありません

解説： enum の値指定で，整数以外の定数で値を指定しています。

List 2-39 ● エラー例 (err039.c)

```
enum foo { R = 1.0, G, B };
```

```
err039.c:        1: Error  : enum 'R' の値が整数定数ではありません
```

XC Ver. 2.10

```
err039.c      1 :Error       21:enum constant error.
```

診断： 列挙型が整数範囲を越えています

解説： enum の値の範囲が整数定数の範囲を逸脱しています。

List 2-40 ● エラー例 (err040.c)

```
enum foo { R = 2147483648, G, B };
```

```
err040.c:        1: Error  : 列挙型が整数範囲を越えています
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 戻り値が不完全です

解説： 構造体／共用体を返す関数を定義するときに，その構造体や共用体が不完全な場合に発生します。

List 2-41 ● エラー例 (err041.c)

```
struct foo;
struct foo bar (void)
{
}
```

```
err041.c: In function bar:
err041.c:       3: Error   : 戻り値が不完全です
```

診断： 引数が2つのスタイルで渡されています

解説： 関数の定義で **ANSI C** の方法と同時に "伝統的な" 方法を用いると発生します。通常は考えられないエラーです。

List 2-42 ● エラー例 (err042.c)

```
int foo (int x)
int x;
{
  return x;
}
```

```
err042.c: In function foo:
err042.c:       2: Error   : 引数が2つのスタイルで渡されています
err042.c:       1: Error   : 'x' が再定義されました
err042.c:       2: Error   : 'x' が再宣言されました
```

XC Ver. 2.10

```
err042.c     2 :Error       22:declaration error.
```

診断： 引数がありません

解説： 関数の定義で `int foo (int) { }` のように引数識別子を忘れてしまうと発生します。宣言ではこの識別子はあってもなくても同じです。

List 2-43 ● エラー例 (err043.c)

```
int foo (int)
{
  return 0;
}
```

```
err043.c: In function foo:
err043.c:         1: Error   : 引数がありません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 引数 '*Ident*' は void と宣言されています

解説： 関数の定義で `int foo (void bar) { }` のように，引数を void と宣言すると発生します。

List 2-44 ● エラー例 (err044.c)

```
int foo (void x)
{
  return 0;
}
```

```
err044.c: In function foo:
err044.c:         1: Error   : 引数 'x' は void と宣言されています
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 引数の名前がありません

解説： **GCC** のソースを見ていろいろやってみましたが，このエラーは発生しませんでした。"伝統的な" 関数定義で発生するはずなのですが…。

診断： 引数 '*Ident*' が複数あります

解説： "伝統的な" 関数定義で，同一名称の引数を複数宣言すると発生します。

List 2-45● エラー例 (err045.c)

```
int foo (x, x)
int x,x;
{
  return 0;
}
```

```
err045.c: In function foo:
err045.c:        2: Error   : 'x' が再定義されました
err045.c:        2: Error   : 'x' が再宣言されました
err045.c:        2: Error   : 引数'x' が複数あります
err045.c:        2: Error   : 'x' が再定義されました
err045.c:        2: Error   : 'x' が再宣言されました
err045.c:        2: Error   : 引数'x' がありません
```

XC Ver. 2.10

```
err045.c     1 :Error      29:identifier redeclaration.
err045.c     2 :Error      29:identifier redeclaration.
```

診断： 引数 '*Ident*' が不完全です

解説： 引数の構造体が不完全です。

List 2-46● エラー例 (err046.c)

```
struct foo;
void bar (struct foo x)
{}
```

```
err046.c:        3: Error   : 引数'x' が不完全です
```

診断： 引数 '*Ident*' がありません

解説： "伝統的な" 関数定義で，関数のパラメータ名称と一致する引数宣言が見つからない場合に発生します。

List 2-47 ● エラー例 (err047.c)

```
int foo (x, y)
int x;
int yy;
{
  return 0;
}
```

```
err047.c: In function foo:
err047.c:       3: Error   : 引数'yy'がありません
```

XC Ver. 2.10

```
err047.c      4 :Error       43:function argument error : yy.
```

診断：　引数の数がプロトタイプと異なります

解説：　引数の数がプロトタイプと一致しません。正しい数に直してください。このエラーは“伝統的な”関数定義で発生します。

List 2-48 ● エラー例 (err048.c)

```
int foo (int, int);
int foo (x)
int x;
{
  return 0;
}
```

```
err048.c: In function foo:
err048.c:       4: Error   : 引数の数がプロトタイプと異なります
```

XC Ver. 2.10

```
err048.c      4 :Error       84:mismatched number of parameters : foo.
```

診断：　引数‘*Ident*’がプロトタイプと異なります

解説：　引数のタイプがプロトタイプと異なっています。両者を一致させてください。

List 2-49 ● エラー例 (err049.c)

```
int foo (double);
int foo (x)
int x;
{
  return 0;
}
```

```
err049.c: In function foo:
err049.c:       4: Error   : 引数'x' がプロトタイプと異なります
```

XC Ver. 2.10

```
err049.c      4 :Warning    16:argument type mismatch.
```

診断： プロトタイプは int です
引数は int のみマッチします

解説： 関数宣言をプロトタイプで行い，関数の定義を"伝統的な"方法で行った場合に発生します。詳しい技術的な内容は Vol. 1 の第 2 章「**X68000 GCC**」で説明してありますが,"伝統的な"関数定義を行う場合，プロトタイプには int より小さいサイズの整数型は引数として宣言できないことを覚えておいてください。

List 2-50 ● エラー例 (err050.c)

```
int foo (char);
int foo (x)
char x;
{
  return x;
}
```

```
err050.c: In function foo:
err050.c:       4: Error   : 引数'x' がプロトタイプと異なります
err050.c:       4: Error   : プロトタイプは int です
err050.c:       4: Error   : 引数は int のみマッチします
```

特記事項

XC Ver. 2.10 および GCC Ver. 2.** ではまったくエラーになりません。**X680x0 GCC** では，このメッセージは出ません。すなわちエラーにはなりません。

診断： プロトタイプは double です
引数は double のみマッチします

解説： 関数宣言をプロトタイプで行い，関数の定義を“伝統的な”方法で行った場合に発生します。詳しい技術的な内容は Vol. 1 第 2 章「**X68000 GCC**」で説明してありますが,“伝統的な”関数定義を行う場合，プロトタイプには double より小さいサイズの浮動小数点型は引数として宣言できないことを覚えておいてください。

List 2-51 ● エラー例 (err051.c)

```
float foo (float);
float foo (x)
float x;
{
  return x;
}
```

```
err051.c: In function foo:
err051.c:       4: Error  : 引数'x'がプロトタイプと異なります
err051.c:       4: Error  : プロトタイプは double です
err051.c:       4: Error  : 引数は double のみマッチします
```

特記事項

XC Ver. 2.10 および GCC Ver. 2.** ではまったくエラーになりません。**X680x0 GCC** では，このメッセージは出ません。すなわちエラーにはなりません。

診断： 空の宣言です

解説： extern ; のような意味のない宣言がこのエラーを引き起こします。

List 2-52 ● エラー例 (err052.c)

```
extern ;
```

```
err052.c:       1: Error  : 空の宣言です
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： '*Ident*' は未宣言です

解説： 変数 *Ident* が未宣言です。C 言語ではすべての変数は宣言の後で使用できます。

List 2-53 ● エラー例 (err053.c)

```
int foo (void)
{
  return x;
}
```

```
err053.c: In function foo:
err053.c:       3: Error  : 'x' は未宣言です
err053.c:       3: Error  : (未宣言識別子は出現した関数ごとに
                            1 度しか報告しません)
```

XC Ver. 2.10

```
err053.c   3 :Error   4:undefined identifier in function name : x .
```

診断： '*Ident*' は不完全です

解説： 構造体の宣言が不完全です。

List 2-54 ● エラー例 (err054.c)

```
struct foo;
struct foo d;
void bar ()
{
  func (d);
}
```

```
err054.c: In function bar:
err054.c:       5: Error  : 'd' は不完全です
err054.c: At top level:
err054.c:       2: Error  : 'd' のサイズは不明です
```

診断： 配列範囲が未定義です

解説： 配列の添字の範囲が未定義です。

List 2-55 ● エラー例 (err055.c)

```
int foo (int ar[][])
{
  return ar[0][0];
}
```

```
err055.c: In function foo:
err055.c:        3: Error   : 配列範囲が未定義です
```

XC Ver. 2.10

```
err055.c       1 :Error       40:element index missing.
err055.c       3 :Error       62:type conversion error.
```

2.1.2　初期化でのエラー

ここでは，変数の初期化についてのエラーを集めました。初期化式は“伝統的な C コンパイラ”と“**ANSI C** コンパイラ”とでは，かなり仕様が異なるので注意してください。

診断：　extern‘*Ident*’は初期化できません

解説：　ブロック内での **extern** 宣言された変数を初期化しています。関数外での **extern** 宣言された変数を初期化するのは違反ではありませんが，ワーニングが発生します。

List 2-56 ● エラー例 (err056.c)

```
int foo (void)
{
  extern int x = 0;
}
```

```
err056.c: In function foo:
err056.c:        3: Warning: ‘x’を‘extern’で初期化しようとしています
err056.c:        3: Error   : extern‘x’は初期化できません
```

XC Ver. 2.10

```
err056.c      3 :Error       22:declaration error.
```

診断： typedef '*Ident*' が変数のように初期化されています

解説： typedef 構文で初期化式を使うと発生します。

List 2-57 ● エラー例 (err057.c)

```
1:  typedef int foo = 10;
```

```
err057.c:       1: Error   : typedef 'foo' が変数のように初期化されていま
す
```

XC Ver. 2.10

```
err057.c      1 :Error       22:declaration error.
err057.c      1 :Error        9:external definition error.
```

診断： 関数 '*Ident*' が変数のように初期化されています

解説： 関数宣言に対して初期化式を用いると発生します。

List 2-58 ● エラー例 (err058.c)

```
1:  int foo (int) = 10;
```

```
err058.c:       1: Error   : 関数'foo' が変数のように初期化されています
```

XC Ver. 2.10

```
err058.c      1 :Error        9:external definition error.
```

診断： 変数 '*Ident*' は初期化できません

解説： 構造体等を不完全なまま初期化すると発生します。

List 2-59 ● エラー例 (err059.c)

```
struct buf;
struct buf x = 10;
```

```
err059.c:       2: Error   : 変数'x'は初期化できません
```

XC Ver. 2.10

```
err059.c      1 :Error       71:structure declaration error.
err059.c      2 :Error       14:undefined struct/union name.
err059.c      2 :Error       54:undefined struct/union used.
err059.c      2 :Error       55:struct/union initializer over.
```

診断：　'*Ident*' の要素が不完全なタイプです

解説：　構造体の配列で，その構造体自身が不完全な場合に，その配列が初期化式をもっていると発生します。

List 2-60 ● エラー例 (err060.c)

```
struct foo;
struct foo bar[1] = { 10 };
```

```
err060.c:       2: Error   : 'bar'の要素が不完全なタイプです
err060.c:       2: Error   : 'bar'のサイズは不明です
```

XC Ver. 2.10

```
err060.c      1 :Error       71:structure declaration error.
err060.c      2 :Error       14:undefined struct/union name.
err060.c      2 :Error       14:undefined struct/union name.
err060.c      2 :Error       54:undefined struct/union used.
err060.c      2 :Error       55:struct/union initializer over.
```

診断：　*Ident* の初期値のサイズが決定できません

解説：　初期化式でエラーが発生した場合に付随して発生するエラーです。

診断： 配列‘*Ident*’の要素数がありません

解説： 初期化式でエラーが発生した場合に付随して発生するエラーです。

診断： 引数‘*Ident*’が初期化されています

解説： 伝統的な関数定義形式で，引数に初期化式をつけて宣言すると発生するエラーです。

List 2-61 ● エラー例 (err061.c)

```
int foo (x)
int x = 10;
{}
```

```
err061.c:        2: Error   : 引数‘x’が初期化されています
```

XC Ver. 2.10

```
err061.c      2 :Error       22:declaration error.
err061.c      2 :Error       56:compound statement error.
err061.c      2 :Warning     15:function return value mismatch.
err061.c      3 :Error        9:external definition error.
```

診断： 初期化式が定数ではありません

解説： 初期化式は関数の内部以外では定数でなければなりません。

List 2-62 ● エラー例 (err062.c)

```
int a,b,c;
int ar[] = {a, b, c};
```

```
err062.c:        2: Error   : 初期化式が定数ではありません
```

XC Ver. 2.10

このエラーメッセージは誤植ではありません。

```
2:Illegal initialization
2:Illegal initialization
2:Illegal initialization
```

診断：　初期化式が複雑すぎます

解説：　まず発生しないエラーです。

診断：　char 配列を広い文字で初期化しています

解説：　char 配列を "L" を先頭にもつ文字列で初期化しています。

List 2-63 ● エラー例 (err063.c)

```
1:  char ar[] = L"幅広文字列";
```

```
err063.c:       1: Error   : char 配列を広い文字で初期化しています
```

診断：　int 配列を短い文字で初期化しています

解説：　int 配列を普通の文字列で初期化しています。

List 2-64 ● エラー例 (err064.c)

```
1:  int ar[] = "普通の文字列";
```

```
err064.c:       1: Error   : int 配列を短い文字で初期化しています
```

診断：　配列を非定数表現で初期化しています

解説：　このエラーを発生させる例を見つけることができませんでした。

診断：　union に初期化すべきメンバがありません

解説：　共用体に初期化すべきメンバがありません。

List 2-65 ● エラー例 (err065.c)

```
union {
} buf = { 10 };
```

```
err065.c:      2: Error   : union に初期化すべきメンバがありません
```

XC Ver. 2.10

```
err065.c     2 :Error      14:undefined struct/union name.
err065.c     2 :Error      55:struct/union initializer over.
```

診断：　初期化要素は 1 つしか必要ありません

解説：　初期化式によけいな式があります。

List 2-66 ● エラー例 (err066.c)

```
int ar[] = { { 0, 1}, 2};
```

```
err066.c:      1: Error   : 初期化要素は 1 つしか必要ありません
```

XC Ver. 2.10

```
err066.c     1 :Error      19:too many initializer.
```

診断：　初期化式に {} は不正です

解説：　{} が必要でない初期化式に {} を使っています。これは **GCC** 拡張の仕様を用いた場合のエラーです。

List 2-67 ● エラー例 (err067.c)

```
typedef struct point
{
  short x;
  short y;
} POINT;
int foo = (POINT){ 0, 1};
```

```
err067.c:        6: Error   : 初期化式に {} は不正です
```

診断：　可変サイズオブジェクトは初期化できません

解説：　可変サイズ配列は初期化できません。

List 2-68 ● エラー例 (err068.c)

```
void
foo (int size)
{
  char ar[size] = { 0, 0};
}
```

```
err068.c: In function foo:
err068.c:        4: Error   : 可変サイズオブジェクトは初期化できません
```

診断：　不正な初期化です

解説：　初期化が正しくありません。

List 2-69 ● エラー例 (err069.c)

```
int ar[] = 20;
```

```
err069.c:        1: Error   : 不正な初期化です
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

2.1.3 型変換，sizeof でのエラー

C 言語では“型変換”がよく起こります。ここではキャスト等の型変換に関するエラーについて説明します。

診断： ポインタに変換できません

解説： float や double，構造体のような，ポインタには変換できないデータをポインタに変換しようとしています。そのようなキャストはできません。

List 2-70 ● エラー例 (err070.c)

```
struct {
  int x;
} buf;
int *
foo ()
{
  return (int *) buf;
}
```

```
err070.c: In function foo:
err070.c:       7: Error   : ポインタに変換できません
```

XC Ver. 2.10

```
err070.c     7 :Error     68:do not same operand size pointer.
```

診断： ポインタは浮動小数点型にできません
浮動小数点型にできません
整数型に変換できません

解説： キャストで，ポインタや構造体を変換できないタイプに変換しようとすると発生します。

List 2-71 ● エラー例 (err071.c)

```
struct point
{
  short x;
  short y;
} buf;

```

```
struct point *points;
double d;
float f;
int i;

void foo ()
{
  d = (double) points;
  f = (float) buf;
  i = (int) buf;
}
```

```
err071.c: In function foo:
err071.c:      14: Error   : ポインタは浮動小数点型にできません
err071.c:      15: Error   : 浮動小数点型にできません
err071.c:      16: Error   : 整数型に変換できません
```

XC Ver. 2.10

```
err071.c     14 :Error      62:type conversion error.
err071.c     14 :Error      62:type conversion error.
err071.c     15 :Error      68:do not same operand size pointer.
err071.c     15 :Error      68:do not same operand size pointer.
err071.c     16 :Error      68:do not same operand size pointer.
err071.c     16 :Error      68:do not same operand size pointer.
```

診断： 構造体には変換できません

解説： キャストの誤りなどで，その構造体には変換できないデータを変換しようとしています。プロトタイプで“*”のつけ忘れでも，引数を構造体に変換しようとして発生します。

List 2-72● エラー例 (err072.c)

```
struct point
{
  short x;
  short y;
} buf;

void foo ()
{
  buf = (struct point) 10;
}
```

```
err072.c: In function foo:
err072.c:       9: Error   : 構造体には変換できません
```

特記事項

XC Ver. 2.10 では，これはエラーになりません。この例では，`move.l #10,_buf` というコードを生成します。

診断： bit-field に 'sizeof' は適用できません

解説： ビットフィールドに sizeof () を使っています。ビットフィールドには sizeof () は使えません。

List 2-73 ● エラー例 (err073.c)

```
struct {
  unsigned bits0: 8;
  unsigned bits1: 8;
} buf;
int foo (void)
{
  return sizeof (buf.bits0);
}
```

```
err073.c: In function foo:
err073.c:       7: Error   : bit-field に'sizeof' は適用できません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

2.1.4 関数記述でのエラー

ここでは，一般的な C 言語で関数内部を記述する際に起こるエラーについて説明します。

診断： void に値はありません

解説： void である関数の戻り値を代入しようとしたり，void へのポインタに * 演算子を用いたりしたときに発生します。void は値も形もないオブジェクトです。

List 2-74 ● エラー例 (err074.c)

```
void bar ();
int
foo ()
{
  return bar ();
}
```

```
err074.c: In function foo:
err074.c:        5: Error    : void に値はありません
```

XC Ver. 2.10

```
err074.c      5 :Error       70:void type error in expression.
```

診断： case ラベルが整数型ではありません

解説： case の後に続くラベルが整数型か列挙型ではありません。

List 2-75 ● エラー例 (err075.c)

```
void foo (int x)
{
  switch (x)
    {
    case 1.0:
      break;
    }
}
```

```
err075.c: In function foo:
err075.c:        5: Error    : case ラベルが整数型ではありません
```

XC Ver. 2.10

```
err075.c      5 :Error       57:constant expression required.
```

診断： case ラベルが switch 文の中にありません

解説： switch 文が作るブロック以外に case ラベルをおくと発生します。

List 2-76 ● エラー例 (err076.c)

```
void foo (int x)
{
  switch (x)
    {
    case 0:
      break;
    }
  case 1:
```

```
        ;
}
```

```
err076.c: In function foo:
err076.c:       8: Error   : case ラベルが switch 文の中にありません
```

XC Ver. 2.10

```
err076.c     8 :Error      48:case statement error.
```

診断： case の値が同値です

解説： case に続くラベルの値が同じ値をとっています。別の値を設定してください。

List 2-77 ● エラー例 (err077.c)

```
void foo (int x)
{
  switch (x)
    {
    case 0:
      break;
    case 0:
      break;
    }
}
```

```
err077.c: In function foo:
err077.c:       7: Error   : case の値が同値です
```

XC Ver. 2.10

このエラーメッセージは誤植ではありません。

```
3:c1 : Duplicate case(0) label(6)
```

診断： default ラベルが switch 文の中にありません

解説： switch 文が作るブロック以外に default ラベルをおくと発生します。

List 2-78 ● エラー例 (err078.c)

```
void foo (int x)
{
  switch (x)
    {
    case 0:
      break;
    }
  default:
    ;
}
```

```
err078.c: In function foo:
err078.c:       8: Error   : default ラベルが switch 文の中にありません
```

XC Ver. 2.10

```
err078.c     8 :Error      49:default statement error.
```

診断： default ラベルが 2 つ以上あります

解説： 1 つの switch 文に 2 つ以上の default ラベルがあります。

List 2-79 ● エラー例 (err079.c)

```
void foo (int x)
{
  switch (x)
    {
    case 0:
      break;
    default:
      break;
    default:
      break;
    }
}
```

```
err079.c: In function foo:
err079.c:       9: Error   : default ラベルが 2 つ以上あります
```

XC Ver. 2.10

```
err079.c     9 :Error      51:more than 1 default.
```

診断： break が loop か switch の中にありません

解説： break がおかしな位置にあります。正しい位置においてください。

List 2-80● エラー例 (err080.c)

```
void foo (int x)
{
  if (x)
    break;
}
```

```
err080.c: In function foo:
err080.c:       4: Error   : break が loop か switch の中にありません
```

XC Ver. 2.10

```
err080.c      4 :Error       33:break error.
```

診断： continue が loop 外部です

解説： continue がおかしな位置にあります。正しい位置においてください。

List 2-81● エラー例 (err081.c)

```
void foo (int x)
{
  if (x)
    continue;
}
```

```
err081.c: In function foo:
err081.c:       4: Error   : continue が loop 外部です
```

XC Ver. 2.10

```
err081.c      4 :Error       34:continue error.
```

診断： 文法違反

解説： シンタックスエラーです。

診断： LASCII 文字列同士は連結できません

解説： LASCII 文字を 2 つ並べての連結はできません。

List 2-82 ● エラー例 (err082.c)

```
char *ch = "\@ABCD" "\@abcd";
```

```
err082.c:       1: Error  : LASCII 文字列同士は連結できません
```

診断： 違法な'\'です

解説： '\' がおかしな位置にあります。正しい位置に直してください[1]。

1)設定によっては，'\'は'¥'とも表示されます。

診断： 浮動小数点表現が違法です

解説： 浮動小数点の表現がまちがっています。

診断： 不適切な浮動小数表現です

解説： 浮動小数点の表現がまちがっています。

診断： 数字表現が違法です

解説： 整数定数表現がまちがっています。

診断： 不適切な数字表現です

解説： 定数表現がまちがっています。

診断： ‘f’が2つ以上あります

解説： 浮動小数点表記の末尾に“f”がよけいについています。

診断： ‘l’が2つ以上あります

解説： 整数定数表記の末尾に“l”がよけいについています。

診断： ‘u’が2つ以上あります

解説： 整数定数表記の末尾に“u”がよけいについています。

診断： ‘l’が3つ以上あります

解説： 整数定数表記の末尾に“l”がよけいについています。

診断： 数字末尾が違法です

解説： 定数表現が誤っています。

診断： 文字列表現が誤っています

解説： 文字列の表現が誤っています。

診断： 文字がありません

解説： 文字定数の中身がありません。

診断： 文字列定数が長すぎます

解説： 通常は発生しないエラーです。

診断： \x の後が 1 6 進表現ではありません

解説： \x に続くシーケンスが正しい定数を表していません。正しい 16 進数表記に直してください。

List 2-83 ● エラー例 (err083.c)

```
char *ch = "\xOP";
```

```
err083.c:       1: Error  : \x の後が 1 6 進表現ではありません
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 不正な void 表現です

解説： 変数を void にキャストして関数に渡したりすると発生します。

List 2-84 ● エラー例 (err084.c)

```
int d;
void foo ()
{
  func ((void)d);
}
```

```
err084.c: In function foo:
err084.c:       4: Error  : 不正な void 表現です
```

特記事項

XC Ver. 2.10 ではまったくエラーになりません。

診断： 左辺値ではありません

解説： メッセージの意味するとおりなのですが，適切なエラー例を作成できませんでした。

診断： struct に '*Ident*' がありません

解説： 構造体のメンバ *Ident* が発見できません。構造体の定義が誤っているか，メンバ名が正しくありません。

List 2-85 ● エラー例 (err085.c)

```
struct foo {
  int xx;
  int yy;
};
int bar (struct foo *p)
{
  return p -> xy;
}
```

```
err085.c: In function bar:
err085.c:      7: Error   : struct に'xy' がありません
```

XC Ver. 2.10

```
err085.c  7 :Error    4:undefined identifier in function name : xy .
err085.c  7 :Error   67:struct reference error.
err085.c  7 :Error   67:struct reference error.
err085.c  7 :Warning 21:pointer type mismatch.
```

診断： union に '*Ident*' がありません

解説： 共用体のメンバ *Ident* が発見できません。共用体の定義が誤っているか，メンバ名が正しくありません。

List 2-86 ● エラー例 (err086.c)

```
union foo {
  int xx;
  int yy;
};
int bar (union foo *p)
{
  return p -> xy;
}
```

```
err086.c: In function bar:
err086.c:       7: Error   : union に'xy' がありません
```

XC Ver. 2.10

```
err086.c  7 :Error    4:undefined identifier in function name : xy .
err086.c  7 :Error   67:struct reference error.
err086.c  7 :Error   67:struct reference error.
err086.c  7 :Warning 21:pointer type mismatch.
```

診断：　*'Ident'* は struct か union のはずです

解説：　構造体／共用体の参照方法が誤っています。

List 2-87 ● エラー例 (err087.c)

```
union foo {
  int xx;
  int yy;
};
int bar (union foo *p)
{
  return p.xx;
}
```

```
err087.c: In function bar:
err087.c:       7: Error   : 'xx' は struct か union のはずです
```

XC Ver. 2.10

```
err087.c     7 :Error      67:struct reference error.
```

診断：　ポインタスケールが不完全です

解説：　不完全な構造体へのポインタを参照すると発生します。

List 2-88 ● エラー例 (err088.c)

```
struct foo;
void
bar (struct foo *p)
{
  foo (*p);
}
```

```
err088.c: In function bar:
err088.c:        5: Error   : ポインタスケールが不完全です
```

診断： '*Operate*' は不正です

解説： 演算 *Operate* は使えません。

List 2-89 ● エラー例 (err089.c)

```
struct foo
{
  int x;
};
int
bar (struct foo x)
{
  x -> x = 2;
}
```

```
err089.c: In function bar:
err089.c:        8: Error   : '->' は不正です
```

XC Ver. 2.10

```
err089.c      8 :Error       61:operand type mismatch.
```

診断： 配列参照に添え字がありません

解説： 配列の参照がまちがっています。

診断：　添え字が整数ではありません

解説：　配列の添字が整数型ではありません。

List 2-90 ● エラー例 (err090.c)

```
int ar[10];
double ind;
int
bar (void)
{
  return ar[ind];
}
```

```
err090.c: In function bar:
err090.c:       6: Error  : 添え字が整数ではありません
```

XC Ver. 2.10

```
err090.c     6 :Error      62:type conversion error.
```

診断：　添え字を持つ変数が配列かポインタではありません

解説：　[ ] が配列／ポインタ以外に使われています。

List 2-91 ● エラー例 (err091.c)

```
int ar;
int ind;
int
bar (void)
{
  return ar[ind];
}
```

```
err091.c: In function bar:
err091.c:       6: Error  : 添え字を持つ変数が配列かポインタではありません
```

XC Ver. 2.10

```
err091.c     6 :Error      61:operand type mismatch.
```

診断： 関数でないオブジェクトを呼ぼうとしています

解説： 関数でもなく関数へのポインタでもない変数を，関数として呼び出そうとしています。

List 2-92● エラー例 (err092.c)

```
int func;
int
bar (void)
{
  return func ();
}
```

```
err092.c: In function bar:
err092.c:       5: Error   : 関数でないオブジェクトを呼ぼうとしています
```

XC Ver. 2.10

```
err092.c      5 :Error       61:operand type mismatch.
err092.c      5 :Error       65:non-function call.
err092.c      5 :Error       61:operand type mismatch.
```

診断： ‘*Ident*’ の引数が多すぎます

解説： プロトタイプと引数の数が一致していません。

List 2-93● エラー例 (err093.c)

```
int func (int);
int
bar (void)
{
  return func (0, 1);
}
```

```
err093.c: In function bar:
err093.c:       5: Error   : ‘func’の引数が多すぎます
```

XC Ver. 2.10

```
err093.c      5 :Error       76:too few parameters in call.
```

診断： 関数の引数が多すぎます

解説： プロトタイプと引数の数が一致していません。

List 2-94 ● エラー例 (err094.c)

```
int (*func) (int);
int
bar (void)
{
  return func (0, 1);
}
```

```
err094.c: In function bar:
err094.c:        5: Error  : 関数の引数が多すぎます
```

XC Ver. 2.10

```
err094.c      5 :Error       76:too few parameters in call.
```

診断： 呼び出し関数の引数が不完全です

解説： 構造体を引数にする関数の場合，引数の構造体は不完全であってはいけません。

List 2-95 ● エラー例 (err095.c)

```
struct foo;
int func (struct foo);
void
bar (void)
{
  func (0);
}
```

```
err095.c:        2: Warning: 引数が不完全です
err095.c: In function bar:
err095.c:        6: Error  : 呼び出し関数の引数が不完全です
```

診断： *'Ident'* の引数が少なすぎます

解説： プロトタイプと引数の数が一致していません。

List 2-96● エラー例 (err096.c)

```
void foo (int, int);
void
bar (void)
{
  foo (1);
}
```

```
err096.c: In function bar:
err096.c:       4: Error   : 'foo' の引数が少なすぎます
```

XC Ver. 2.10

```
err096.c     4 :Error      76:too few parameters in call.
```

診断： 引数が少なすぎます

解説： プロトタイプと引数の数が一致していません。

List 2-97● エラー例 (err097.c)

```
void (*foo) (int, int);
void
bar (void)
{
  foo (1);
}
```

```
err097.c: In function bar:
err097 .c:       4: Error   : 引数が少なすぎます
```

XC Ver. 2.10

```
err097.c     4 :Error      76:too few parameters in call.
```

診断： 不当な表現です

解説： 演算子の使い方が誤っています。

診断： 演算 ‘*Operator*’ オペランドが不正です

解説： ポインタを乗算したりするような不正な演算を行うと発生します。

List 2-98 ● エラー例 (err098.c)

```
int *x;
void foo ()
{
  x *= 2;
}
```

```
err098.c: In function foo:
err098.c:       4: Error   : 演算‘*’ オペランドが不正です
```

XC Ver. 2.10

```
err098.c      4 :Error      82:illegal use of pointer.
```

診断： bit-field のメンバ ‘*Ident*’ のアドレスは参照できません

解説： ビットフィールドに “&” 演算子を使っています。

List 2-99 ● エラー例 (err099.c)

```
struct {
  unsigned bit0: 8;
  unsigned bit1: 8;
  unsigned bit2: 8;
  unsigned bit3: 8;
} buf;
void *
foo ()
{
  return (void *) &buf.bit1;
}
```

```
err099.c: In function foo:
err099.c:      10: Error   : bit-field のメンバ‘bit1’ のアドレスは参照でき
ません
```

XC Ver. 2.10

```
err099.c     10 :Error       66:no left value.
```

診断： 条件式での type が異なっています

解説： 条件式で代入できないタイプを代入しています。

List 2-100 ● エラー例 (err100.c)

```
typedef struct
{
  int x;
} BUF;
BUF xx;
int x;
foo ()
{
  x  = x ? x : xx;
}
```

```
err100.c: In function foo:
err100.c:       9: Error   : 条件式での type が異なっています
```

XC Ver. 2.10

```
err100.c      9 :Error       68:do not same operand size pointer.
err100.c      9 :Warning     20:type mismatch.
```

診断： 配列にキャストできません

解説： 配列にはキャストできません。

List 2-101 ● エラー例 (err101.c)

```
int
foo (int x)
{
  return ((int []) x)[0];
}
```

```
err101.c: In function foo:
err101.c:       4: Error   : 配列にキャストできません
err101.c:       4: Error   : 添え字を持つ変数が配列かポインタではありません
```

XC Ver. 2.10

```
err101.c      4 :Error      63:type conversion illegal.
```

診断： *Action* は異なったタイプです

解説： *Action* には "代入" や "引数変換" 等の処理が入ります。異なったタイプを代入したり引数で渡そうとしたりしています。

List 2-102 ● エラー例 (err102.c)

```
typedef struct
{
  int x;
} INT;
void foo (int);
INT X;
int x;
void
bar (void)
{
  x = X;
  foo (X);
}
```

```
err102.c: In function bar:
err102.c:      11: Error   : 代入は異なったタイプです
err102.c:      12: Error   : 関数'foo' 1 番目の引数は異なったタイプです
```

XC Ver. 2.10

```
err102.c     11 :Error      68:do not same operand size pointer.
err102.c     12 :Warning    14:struct/union for function argument.
err102.c     12 :Warning    16:argument type mismatch.
```

診断：　switch 文の条件が整数ではありません

解説：　switch 文の条件式が整数型ではないタイプが指定されています。

List 2-103 ● エラー例 (err103.c)

```
int
foo (double x)
{
  switch (x)
  {
  case 0:
    break;
  }
}
```

```
err103.c: In function foo:
err103.c:       4: Error   : switch 文の条件が整数ではありません
```

XC Ver. 2.10

```
err103.c     4 :Error      61:operand type mismatch.
```

2.1.5 GCC 拡張でのエラー

ここでは GNU による **GCC** 拡張，および **X68000 GCC** の独自拡張によって発生するエラーについて説明します。

診断：　'{}' 表現は関数内部だけで使用できます

解説：　関数の外で { } で囲まれたブロックを使っています。ブロックは関数内部でだけ使えます。

List 2-104 ● エラー例 (err104.c)

```
int foo = ({ int y; y;});
```

```
err104.c:       1: Error   : '{}' 表現は関数内部だけで使用できます
err104.c:       1: Error   : 文法違反　文字 '{'
```

診断：　不正な#pragma dump です

解説：　#pragma dump の使い方が誤っています。

診断：　Dump ファイルを作れません *Filename*

解説：　*Filename* の Dump ファイルを開くことができません。

診断：　Dump ファイル書出しに失敗しました

解説：　なんらかのエラーによって Dump ファイルの作成に失敗しました。ディスクの容量等を調べてください。

診断：　LASCII 文字列には wide 文字は使えません

解説：　LASCII 文字列に接頭子 “L” を指定しています。

診断：　LASCII 文字列は 255 文字までです

解説：　LASCII 文字列の長さが 255 文字を超えています。255 文字以内にしてください。

診断：　SXCALL 関数のアドレスはとれません

解説：　SXCALL 属性の関数のアドレスを求めています。

List 2-105 ● エラー例 (err105.c)

```
SXCALL int bar (int *);
void
foo ()
{
  int (*func)() = bar;
  func ();
}
```

```
err105.c: In function foo:
err105.c:       5: Error   : SXCALL 関数のアドレスはとれません
```

診断： オペランドが複数存在しています
オペランドが矛盾しています
output オペランドに不正な '+' があります
output オペランドに '=' がありません
operand は最大 8 個です
'*CHAR*' が不正な位置にあります

解説： asm 文で発生するエラーです。これらのエラーの詳細を知るには **GCC** の内部と asm 文についての知識が必要です。

診断： 関数の外で可変サイズ変数は使えません

解説： 関数の外で可変サイズの配列を使おうとしています。可変サイズの配列は，関数内部のブロックの先頭でのみ使えます。

List 2-106 ● エラー例 (err106.c)

```
int a;
int ar[a];
```

```
err106.c:        2: Error  : 関数の外で可変サイズ変数は使えません
```

診断： 関数 '*Ident*' は定義できません

解説： DOSCALL, SXCALL 属性の関数を定義しようとしています。この属性の関数は定義できません。

List 2-107 ● エラー例 (err107.c)

```
DOSCALL int
foo ()
{
}
```

```
err107.c: In function foo:
err107.c:        3: Error  : 関数'foo' は定義できません
```

診断： 不正なレジスタ '*RegName*' がレジスタ変数に指定されています

解説： *RegName* のレジスタが存在しないレジスタか，使えないレジスタです。

診断： *Ident* がレジスタ変数ではありません

解説： レジスタ指定を行う変数 *Ident* は `register` 宣言してください。

診断： スタックが不足です
最低追加必要量 *Num* バイトです

解説： コンパイラが使うスタックが不足しました。オプション`-cc1-stack` でスタックを増やしてください。最低追加量は目安でしかありません。なおデフォルトでは，コンパイラのスタックは約 190K バイト確保されています。

診断： フレームポインタレジスタが不正です

解説： フレームポインタに指定されたレジスタは，フレームポインタにはできないレジスタです。

診断： レジスタ変数 '*Ident*' は 68881 指定が必要です

解説： 68881 レジスタをレジスタ指定変数にするには，`-m68881` オプションが必須です。

診断： '*Ident*' は `register` 宣言できないタイプです
`frame pointer` にできないタイプです

解説： レジスタ指定変数は整数型変数だけです。

診断： `global register` 変数は初期化できません

解説： 大域レジスタ変数を初期化すると発生します。

診断： **global register 変数は関数定義の後に宣言できません**

解説： 関数を定義した後で，大域レジスタ変数は宣言できません。

診断： **"a5" は使えません**

解説： SX–Window 開発モードにおいて a5 レジスタは，レジスタ指定変数にはできません。

診断： **relocate は初期化できません**

解説： relocate 記憶クラスは初期化できません。

診断： **relocate は関数内部で宣言できません**

解説： relocate は関数内部で宣言はできません。これは relocate 記憶クラスが特殊な記憶クラスだからです。

2.1.6 宣言に関するワーニング

ここでは宣言関係のワーニングを集めてあります。C 言語では，宣言されていない関数を呼び出すと暗黙の宣言になります。また，関数の実際の処理を宣言なく記述することは，暗黙の宣言と同時に関数を定義することにもなります。このことを頭にいれておくと，ワーニングの意味がよくわかるようになります。

診断： **static'*Ident*' が暗黙に extern に宣言されています**

解説： 関数を宣言しないで[2]呼び出した後，static で定義すると，このワーニングが発生します。**X68000 GCC** ではこのワーニングを発生させないように，static 関数はあらかじめ宣言しておいたほうがコードサイズ上有利です。

2) 暗黙の宣言になります。

診断： '*Ident*' が前に `int` を返すと暗黙に宣言されています

解説： 関数を宣言なく呼び出した場合には“暗黙の宣言”になります。この場合，戻り値は `int`，引数はノーチェックとなります。しかし呼び出し後に，この暗黙宣言と異なった関数定義をすると，このワーニングが発生します。

診断： '*Ident*' は暗黙に関数として宣言されています

解説： *Ident* を暗黙に関数として使った後，同じ *Ident* を変数として宣言しています。これは **X68000** では 100% 暴走しますので，ワーニングよりはエラーに極めて近いワーニングです。

診断： 局所宣言 '*Ident*' が外部と一致しません

解説： { } で囲まれたブロック内部の宣言とトップレベルでの宣言が異なっています。

診断： '*Ident*' の宣言は引数を隠します

解説： *Ident* を関数の引数として使った場合に，その関数内で *Ident* を別に宣言すると発生するワーニングです。当然変数の値は引数ではなく，その変数は未初期化なので，ゴミの値が入っています。もし変数 *Ident* がポインタならば，バスエラーかアドレスエラーになるでしょう。

診断： '*Ident*' の宣言は `local` 変数を隠します

解説： { } で囲まれたブロックで，変数 *Ident* を `extern` 宣言した場合に変数 *Ident* がローカル変数として使われていると，その変数はそのブロック内では参照できなくなります。このワーニングはそのような場合に発生します。

診断： '*Ident*' が暗黙に宣言されました

解説： 関数の呼び出しによる暗黙の関数宣言が行われるたびに，このワーニングが発生します。

診断： 意味のない **keyword,type name** が宣言にあります

解説： 文法上まちがってはいませんが，意味上ではまったく用をなさない語句がプログラムにあります。たいていの場合は，文法エラー寸前の誤った記述です。

診断： '*Ident*' が複数あります

解説： `short short x;` のような宣言で，*Ident* の部分が "`short`" に置き換えられた形で発生するワーニングです。

診断： **long long long** は長すぎます

解説： **GCC** の最大ビット長整数である `long long int` にさらに `long` をつけると発生するワーニングです。

診断： 二重の **const** です
二重の **volatile** です

解説： 同一識別子に `const`，または `volatile` が複数指定されています。

診断： '*Ident*' を '**extern**' で初期化しようとしています

解説： `extern` を付加した識別子を初期化しています。もしこれがヘッダに存在していたら，リンカで重複シンボルエラーになるでしょう。

診断： ANSI C ではサイズ 0 の配列 '*Ident*' は使えません

解説： **ANSI C** では許されていないサイズが 0 の配列を使っています。

診断： ANSI Cでは可変サイズ配列 '*Ident*' は使えません

解説： **ANSI C**では許されていないサイズが可変の配列を使っています。

診断： ANSIでは const,volatile 関数は使えません
ANSI Cでは const や volatile である関数は違法です

解説： **ANSI C**では，関数に const または volatile を指定することはできません。

診断： 関数 'main' は inline にできません

解説： main () 関数を inline 関数にしようとしています。

診断： 可変長引数関数は inline にできません

解説： 可変長引数関数は inline 関数にはできません。

診断： 変数 '*Ident*' が inline 宣言です

解説： 変数に inline が指定されています。**GCC**はこの inline を無視します。

診断： 関数宣言が prototype ではありません

解説： 関数宣言が **ANSI C**の厳密なプロトタイプではありません。

診断： タイプの指定のない引数の名前が関数宣言に現れました

解説： 関数のプロトタイプ宣言で int や double のようなタイプ名が指定されていない変数名が関数引数の部分にあります。

診断： *Ident* タグ名 ‘*Ident*’ が引数として宣言されました
スコープは宣言か定義の中だけです

解説： 関数宣言の引数で，構造体か共用体のタグ名が初めて宣言された場合に発生するワーニングです。

診断： ANSI C では enum の前方参照はできません

解説： enum にタグ名称だけを宣言して，後で実体を定義することは **ANSI C** ではできません。

診断： union が内部宣言です

解説： union を関数宣言の引数で宣言すると発生します。

診断： union のメンバがありません

解説： union にそれを構成するメンバがありません。

診断： bit-field ‘*Ident*’ は ANSI C では不当です

解説： ビットフィールドのタイプは int か unsigned が **ANSI C** の規定です。

診断： bit-field ‘*Ident*’ の幅は負にできません

解説： ビットフィールドの幅が負の数になっています。

診断： enum が内部宣言されました

解説： 関数宣言の引数で enum のタグ名が初めて宣言された場合に発生するワーニングです。

診断： type or storage class がありません

解説： Ident; のようにタイプ指定のない宣言に出会うたびに，このワーニングが発生します。

診断： ‘,’ が enum list の終わりにあります
‘;’ が struct か union の終わりにありません
‘;’ が struct か union に余計にあります

解説： 冗長な “;”，“,” あるいは不足している “;” に対するワーニングです。

診断： ANSI C ではメンバのない宣言は違法です

解説： union，struct にメンバがない宣言は **ANSI C** では違法です。

2.1.7 関数記述でのワーニング

ここでは，一般的な C 言語で起こる関数内部を記述する際に発生するワーニングについて説明します。

診断： label *Label* が参照されていません

解説： *Label* が goto 文で参照されていません。特にこれが原因での弊害はないでしょうが，ムダなラベル宣言であるのは事実です。

診断： 戻り値は int です

解説： 戻り値を宣言しない形式で関数を定義すると，デフォルトで int を返す関数になります。定義ごとにこのワーニングが発生します。

診断： ‘*Ident*’ は ‘int’ です

解説： 伝統的な関数定義で，引数のタイプを宣言しなかった場合にそのタイプ宣言のない引数ごとにこのワーニングが発生します。

診断： volatile 関数は戻ってきません

解説： GCC では，volatile 宣言された関数は呼び出し元へ戻らないことを前提にしています。

診断： non-void 関数が値を返していません

解説： 暗黙宣言，明示宣言どちらの場合でも，void と宣言された関数で値をもった return がない場合に発生します。

診断： 関数が値を返す場合、そうでない場合があります

解説： void でない関数が値を明示して返す場合と，そうでない場合があります。このように値を返さない経路をプログラムが通るとき，暴走する危険性があります。

診断： ANSI C では関数の外の‘;’は違法です

解説： 関数の外によけいな“;”があります。このよけいな“;”は，ほとんどのコンパイラでは無視されます。

診断： ANSI C では空の {} での初期化は違法です

解説： ANSI C では中身のない“{}”での初期化は違法です。

診断： \ x の後が１６進表現ではありません

解説： エスケープシーケンス \x の後に続く文字列が 16 進数を表していない場合に発生します。このワーニングが発生した場合の結果は不定なので，ほとんどエラーと同意に扱うべきワーニングです。

診断： １６進表現が範囲を越えています

解説： 16 進数を処理した結果がオーバーフローした場合に発生するワーニングです。このワーニングが発生した場合の結果は不定なので，エラーと同意に扱うべきワーニングです。

診断： 未知のエスケープシーケンスです

解説： “\” に続く文字が **GCC** の受け入れるエスケープ文字ではありません。このワーニングが発生した場合の結果は不定なので，エラーと同意に扱うべきワーニングです。

診断： 整数定数が範囲を越えました

解説： 整数定数を処理した結果がオーバーフローしました。

診断： `ANSI C` は `newline` で文字定数を連結できません

解説： “ " ” で囲まれた文字列の途中で改行しています。これは **ANSI C** では認められていません。

診断： 多文字文字定数です

解説： `'NASI'` のような複数の文字で構成される文字定数を使うと発生します。

診断： `sizeof` が関数に使われています

解説： `sizeof` 演算子を関数である識別子に対して用いています。

診断： `sizeof` が `void type` に使われています

解説： `sizeof` 演算子を `void` である識別子に対して用いています。

診断： ANSI C は非左辺値配列の添え字を許しません

解説： List 2-108 のような左辺値ではない表現に “[ ]” を用いた場合に発生します。

List 2-108 ● 非左辺値配列参照

```
struct foo
{
 int x[5];
} X;
struct foo func ();
int
bar (int ind)
{
  return func ().x[ind];
}
```

診断： ANSI C は void *と関数ポインタの比較を許しません
異種ポインタを比較しています
ポインタと整数を比較しています
キャストなしで異なるポインタを比較しました
ANSI C はポインタと関数を比較することを許していません
ANSI C ではポインタと関数へのポインタを比較できません
ANSI C は void *と関数へのポインタとの条件式は許していません
ポインタを整数 0 と大小比較しています
条件式のポインタが異種です

解説： ポインタの比較にともなうワーニングです。これらのワーニングが発生する場合，**X68000** ではほぼ期待する動作をしますが，8086 系列を CPU とするマシンでは正しい動作は期待できません。

診断： void ポインタを算術演算しています
関数へのポインタを算術演算しています
void ポインタを減算に用いています
関数へのポインタを減算に用いています

解説： この 4 つのワーニングが発生するソースは，**GCC** 以外ではコンパイルできません。**GCC** ではこれらのポインタは char * と同じ扱いなので，ワーニングになり，エラーにはなりません。

診断： 条件式が何時も 0 です
条件式が何時も 1 です

解説： `unsigned int` を負の数と大小比較するような，最初から結果が明白な比較において発生します。

診断： ++は適用できません
−−は適用できません

解説： `void *` と関数へのポインタに ++，−− を用いた場合に発生します。

診断： '*Ident*' はリードオンリーです

解説： `const` である *Ident* に書き込みを行っています。

診断： 条件式で定数を代入しています
条件式で論理演算結果を代入しています

解説： この 2 つのワーニングは "`if` (条件)" 等での演算子の優先順位まちがい，あるいは "`==`" と "`=`" とのタイプミスによるまちがい等で，条件式が予期していない結果になることをワーニングします。

診断： レジスタ変数 '*Ident*' のアドレスを求めています

解説： `register` 宣言された変数に `&` 演算子を用いると発生します。

診断： 条件式に `pointer/integer` 双方存在します

解説： 3 項演算子 "?" で代入する変数または値が，整数の場合とポインタの場合とが存在するときに発生します。

診断： ANSI C は構造体を構造体にキャストすることは許していません

解説： ある構造体を同じ種類の構造体にキャストして代入しています。別の種類にキャストしての代入はエラーです。

診断： 変換タイプが'volatile'です
変換タイプが'const'です

解説： ポインタどうしの代入で const，volatile でないポインタに const，volatile ポインタを代入する場合に発生します。

診断： *Action* は異種のポインタです
Action で const ポインタをコピーしています
Action で volatile ポインタをコピーしています
Action は異種のポインタです
Action で整数からポインタにしました
Action でポインタから整数にしました

解説： *Action* には代入や引数変換等の行為が入り，ワーニングとして出力されます。これらのワーニングはときと場合によって暴走につながるものと，無視できるものとに分けることができます。

診断： struct,union でない初期化式に'{}'があります

解説： "{ }" が必要のない初期化式に "{ }" が使われています。

診断： 初期化要素が多すぎます

解説： 配列のサイズ等より，初期化する値の数のほうが多い場合に発生するワーニングです。

診断： volatile 宣言された関数に return があります

解説： volatile 宣言した関数は呼び出し元に復帰しないことが前提です。このような関数に return 文が存在すると，GCC はワーニングを出力します。

診断： void でない関数に値のない return があります

解説： void でない関数が値を返していないので，場合によってはコンパイルされたプログラムが暴走します。

診断： void である関数に値を持った return があります

解説： void と宣言した関数に値を返す return があります。直接暴走につながることはまず考えられませんが，美しいプログラムではないでしょう。

診断： 意味のない文です
結果が使用されていません

解説： このワーニングが出ることはめったにありませんが，ワーニングはそのステートメントが何の副作用も起こさない，存在自体が意味がない行為を行っている場合に発生します。

診断： '*Ident*' が未使用です

解説： 変数 *Ident* が使われていません。このワーニングが発生しても暴走等につながることはありません。

診断： switch で列挙値 '*Ident*' の処理がありません

解説： switch 文で enum を扱う際，その列挙値がとるであろう値を網羅していない場合に，扱われていない列挙値ごとにワーニングが発生します。

診断： case 値 'Value' は enum'*Ident*' に存在していません

解説： switch 文で enum 値を case で使わずに直接整数値を記述した場合，その値が列挙に存在しないと発生します。

診断： '*Ident*' が初期化されずに使用されているようです

解説： プログラム上で，値が代入されないまま使われている変数 *Ident* が存在すると発生します。

診断： ‘*Ident*’ は ‘`longjmp`’ で破壊される可能性があります

解説： **ANSI C** では `longjmp ()` 関数で保存復帰される変数は，レジスタ変数と `volatile` 宣言された変数だけです。

診断： ‘*Ident*’ は未使用の引き数です

解説： 引数 *Ident* が関数で使われていません。未使用の引数は呼び出し側のオーバーヘッドを招いて，結果的に実行速度が低下します。

診断： ‘*Ident*’ が宣言されて定義されませんでした

解説： `static` 関数 *Ident* は宣言だけで定義がありません。リンカでエラーになる可能性があります。

診断： ‘*Ident*’ は定義されて使用されませんでした

解説： `static` 関数 *Ident* が，定義されているにもかかわらず呼び出しがありません。ムダなコードが存在することになります。

診断： ‘*Ident0*’ と ‘*Ident1*’ は非常に類似した識別子です

解説： *Ident0* と *Ident1* はコンパイルオプションで指定された先頭文字数が一致しています。**ANSI C** では完全にポータブルな識別子は，大文字小文字の区別なく先頭 8 文字までとなっています。

2.1.8 GCC 拡張でのワーニング

ここではGNU による **GCC** 拡張，および **X68000 GCC** の独自拡張によって発生するワーニングについて説明します。

診断： `ANSI C` では `constructor` 表記はできません
`ANSI C` では '{}' 表現は使用出来ません

解説： **GCC** で拡張されている "{ }" で囲まれた形式の拡張構文は **ANSI C** では認められていません。

診断： `ANSI C` では漢字は識別子に使えません

解説： **X68000 GCC** で拡張されている日本語の識別子は **ANSI C** では使えません。

診断： ＸＣ拡張表現です

解説： `\xb001` のような 2 進数ビット表現は，XC と **X68000 GCC** だけの拡張です。

診断： `ANSI C` では `long long integer` は違法です

解説： **ANSI C** には 64 ビット整数 `long long int` は存在しません。

診断： 漢字を 'L' なく定数に使っています

解説： '漢' のような漢字コード定数は接頭子 "L" が必要です。このデータは `short` より長いビット長の整数に代入しないとオーバーフローします。

診断： `ANSI C` は '? :' 表現を省略できません

解説： **GCC** 拡張の中間項省略条件式を用いると発生します。

診断： 割り込み関数は inline にできません

解説： X68000 GCC 拡張機能の割り込み処理関数を inline にしようとしています。この inline は無視されます。

診断： 可変長引数関数は inline にできません

解説： 可変長引数関数は引数の数が一定ではありません。このような関数は inline にできません。

診断： 関数で破壊されるレジスタが global レジスタ変数に使われています

解説： GCC はこのようなレジスタの破壊を避けるコードは生成できません。

診断： 未知のレジスタです: *RegName*

解説： *RegName* は GCC が知らないレジスタ表記です。

診断： コード最適化は行われていません

解説： 最適化オプションが指定されていません。X68000 では，GCC で作成されていながら最適化を行っていないソフトウェアが多数存在するので，このワーニングを追加してあります。

診断： ANSI C では register 名指定変数は使えません

解説： ANSI C では GCC 拡張のレジスタ指定変数は使えません。

診断： global register が関数内で再使用されています

解説： 大域レジスタ変数が関数内で使われています。結果的にグローバル変数を破壊することになります。

2.2 ………………………… アセンブラのメッセージ

ソースファイルをアセンブルする際に発生したさまざまなエラーに対して，アセンブラは次のようなエラーメッセージを表示します。

```
source.s        1:  bad opecode error
   ↑            ↑          ↑
ファイル名     行番号    エラーメッセージ
```

2.2.1 アセンブルを中断するエラーメッセージ

以下のエラーメッセージは，アセンブル作業を継続できないほどの重大なエラーが発生したときに表示されます。その際，アセンブラはアセンブル作業を中断します。

- `Abort: Out of memory`
  メモリ容量が不足しました。
  「"-m" オプションによってシンボル数を減らす」,「常駐物を解除してメモリを確保する」等の処置を行ってください。
- `Abort: Device full`
  ディスク容量が不足しました。
  十分に空き容量のあるディスク上でアセンブル作業を行ってください。
- `Abort: Too many symbols` [1]
  ソースファイル中のシンボル数が多すぎます。
  "-m" オプションによって最大シンボル数を増やしてください。
- *file name* `file open error`
  ソースファイル *file name* がオープンできません。ソースファイル名指定が誤っています。
- `temporary file open error`
  テンポラリファイルがオープンできません。
  環境変数 `temp` または "-t" オプションによって，正しいテンポラリパスを指定してください。

1)X680x0 HAS ではシンボル数の制限がなくなったため，このエラーメッセージは発生しません。かわりに，外部シンボル数の制限を超えた場合に "Too many external symbols" を表示します。

2.2.2 通常のエラーメッセージ

以下のエラーメッセージは，ソースファイル中の通常のエラーに対して出力されます。

- `Forced error by fail directive`
  `.fail` 疑似命令のエラー発生条件が成立しました。
- `bad opecode error`
  命令か疑似命令が誤っています。
- `division by zero error`
  0 による除算が行われました。
- `expression error`
  式が不適当です。
- `feature not available error`
  表記はサポートされていません。
- `file not found error`
  インクルードファイルがオープンできません。
- `illegal addressing error`
  命令に使用できないアドレス形式です。
- `illegal operand error`
  オペランド指定に誤りがあります。
- `illegal quick size error`
  値がクイックイミディエイトの範囲外です。
- `illegal relative error`
  相対値データが 16／8 ビットオフセットの範囲外です。
- `illegal shift count error`
  シフト／ローテート命令のシフト数が不適当です。
- `illegal size error`
  命令に使用できないサイズ指定です。
- `illegal symbol error`
  シンボル名がレジスタ名と重複しています。
- `illegal value error`
  式の結果が範囲外です。
- `macro nesting over error`
  マクロのネスティングが深すぎます。
- `missing if error`
  条件アセンブル疑似命令のネストが不適当です。
- `missing macro error`
  マクロ定義のネストが不適当です。

- **no symbol error**
  シンボルの必要な疑似命令にシンボルがありません。
- **overflow error**
  式の結果がオーバーフローしました。
- **redefinition error**
  同名のシンボルが再定義されています。
- **register error**
  レジスタ名が不適当です。
- **register size error**
  レジスタサイズが不適当です。
- **too many include file error**
  インクルードファイルのネストが深すぎます。
- **undefined symbol error**
  シンボルが定義も外部参照宣言もされていません。

2.2.3 ワーニングメッセージ

ソースファイル中の致命的でないエラーに対しては，以下のワーニングメッセージが出力されます。

❖ レベル 1 ワーニング

- **Warning: illegal register list**
  movem 命令のレジスタリストの指定順序が誤っています。指定ミスの可能性が大きいので，このワーニングが出力されたらソースファイルを修正するべきです。
- **Warning: terminator not found**
  文字列の引用符が閉じていません。これもソースファイルを修正するべきです。
- **Warning: illegal alignment**
  ワードデータ／命令が奇数アドレスからおかれています。よほど特殊な目的でないかぎり，そのまま実行させるとアドレスエラーになる可能性が大きいです。

❖ レベル 2 ワーニング

- **Warning: HAS expanded specifications**[2)]
  拡張子が“.s”であるソースファイルに対して，**HAS** オリジナルの機能を使用しました。このワーニングが出力されるようなソースプログラムは，純正アセンブラではアセンブルできませんので注意してください。

2)X680x0 HAS では，HAS オリジナルの機能を使用してもこのワーニングは発生しません。

❖ レベル 3 ワーニング

- **`Warning:  short addressing`**
  アドレスレジスタをワード単位で扱っています。アドレスレジスタを，データレジスタの代わりとして使うような場合に出力されることがあります。
- **`Warning:  illegal short value`**
  ディスプレイスメントつきアドレスレジスタ間接形式で **$8000** ～ **$FFFF** の値が使用されるなど，値の誤っている可能性のあるデータ指定がされています。

❖ レベル 4 ワーニング

- **`Warning:  absolute addressing`**
  絶対ロング形式で定数データが指定されています。
- **`Warning:  absolute short addressing`**
  絶対ショート形式で定数データが指定されています。

2.3 ……………………………… HLKのメッセージ

ソースファイルをリンクする際に発生したさまざまなエラーに対して，**HLK**は次のようなエラーメッセージを表示します。

- `Duplicate definition :` *symbol name*
  複数のオブジェクトファイルで，同名のシンボルが外部定義されていた場合に発生します。*symbol name* が外部定義されているオブジェクトファイル名もレポートします。
- `Already read :` *file name*
  指定されたファイルがすでに読み込まれている場合に発生します。
- `Bad option :` *option name*
  指定されたオプションの引数が正しくない場合に発生します。
- `Calc stack over flow in` *xxxx*
  シンボルの演算を行おうとしたときに，ワークが足りなかった場合に発生します。普通に使用していれば，まずお目にかからないと思います。
- `Calc stack under flow in` *xxxx*
  シンボルの演算用のワークに，何もないのに参照しようとした場合に発生します。**HLK**のバグか，アセンブラのバグか，オブジェクトファイルが壊れている可能性があります。
- `Can't open file :` *file name*
  ファイルがオープンできない場合に発生します。
- `Device full :` *file name*
  ディスクが一杯で，ファイルへの書き込みができない場合に発生します。
- `Division by zero in` *xxxx*
  シンボルの演算で0で除算しようとした場合に発生します。
- `File I/O error :` *file name*
  ファイルの読み込み，書き込み時にエラーが発生した場合に発生します。
- `Illegal expression in` *xxxx*
  アドレスを示す属性[1)]のシンボルどうしを，掛けたり足したりした場合に発生します。
- `Illegal file size :` *file name*
  オブジェクトファイルのサイズが奇数バイトの場合に発生します。オブジェクトファイルのサイズは，必ず偶数になっていなければいけません。

1)属性については，Vol. 1 第4章「HLK」を参照してください。

- `Illegal SCD information in` *xxxx*

  解析されていないシンボリックデバッグ情報に遭遇した場合か，オブジェクトファイルの内容が破壊されている場合に発生します。

- `Indirect mode error`

  インダイレクトファイルを複数指定したり，インダイレクトファイル内でインダイレクトファイルを指定すると発生します。

- `Internal error at :` *xxxx*

  内部エラー。**HLK** のバグが原因です。

- `Mismatch roffset size (?_?) !`

  内部エラー。**HLK** のバグが原因です。

- `Not found :` *file name*

  指定されたオブジェクトファイルが見つからない場合に発生します。

- `Not found indirect file :` *file name*

  指定されたインダイレクトファイルが見つからなかった場合に発生します。

- `Not obj, arc file :` *file name*

  指定されたファイルが，オブジェクトファイルでもアーカイブファイルでもない場合に発生します。

- `Out of memory !! (°_°;`

  メモリが不足した場合か，Z ファイル用のオブジェクトファイルをリンクした場合に発生します。Z ファイル用のオブジェクトファイルはリンクできないので，LK を使用してください。メモリが不足している場合は，常駐プログラムを減らしたり，デバイスドライバを削除して，空きメモリを増やして再実行してください。

- `Over flow in` *xxxx*

  シンボルの値を書き込もうとしたときに，バイトもしくはワードの値の範囲を越えた場合に発生します。このときは，オブジェクトファイルのどのセクションのどのアドレス (16 進数) で発生したかの情報をレポートします。

- `Relative error in` *xxxx*

  相対アドレッシングで届かないアドレスを指定した場合や，アドレスを示す属性 のシンボルをワードやバイトの値として書き込もうとすると発生します。このときはオブジェクトファイルの，どのセクションのどのアドレス (16 進数) で発生したかの情報をレポートします。

- `Too many arguments`

  コマンドラインの引数の数が 256 個をオーバーした場合に発生します。

- `Undefined environment variable 'lib'`

  "`-l`" オプションの指定時に，環境変数 `lib` が設定されていない場合に発生します。

- `Undefined symbol(s) in` *file name*

  オブジェクトファイル内で，外部参照しているシンボルが見つからなかった場合に発生します。見つからなかったシンボルもレポートされます。

- **`Unknown option :`** *option name*
  指定されたオプションが **HLK** で用意されていない場合に発生します。
- **`Unknown command :`** *xxxx*
  解析されていないオブジェクトコマンドに遭遇した場合か，オブジェクトファイルの内容が破壊されている場合に発生します。
- **`Warning, duplicate definition :`** *symbol name*
  コモンラベルと普通のシンボルが衝突した場合に発生します。“**`-w`**” オプションを使用すると，発生しなくなります。

Chapter 3

GDBのコマンド

本章では，GDB がサポートするコマンドと行編集のキー操作について説明します。

3.1 ································ コマンドリファレンス

GDB が使用するコマンドを大別すると，次のように 9 つのタイプに分類できます。ここでは，これらのコマンドをリファレンス形式 (アルファベット順) で説明していきます。

- **実行を制御するコマンド (running)** [1]
- **スタックフレームを調査するコマンド (stack)** [2]
- **データに関するコマンド (data)** [3]
- **ブレークポイントに関するコマンド (breakpoints)** [4]
- **ファイルに関するコマンド (files)** [5]
- **プログラムの状態を調査するコマンド (status)** [6]
- **シンボルテーブルを調査するコマンド (obscure)** [7]
- `info` **コマンドのサブコマンド (info)** [8]
- **GDB を設定するコマンド (support)** [9]

1)第3.1.1節参照 (P.181)
2)第3.1.2節参照 (P.186)
3)第3.1.3節参照 (P.188)
4)第3.1.4節参照 (P.193)
5)第3.1.5節参照 (P.196)
6)第3.1.6節参照 (P.199)
7)第3.1.7節参照 (P.200)
8)第3.1.8節参照 (P.201)
9)第3.1.9節参照 (P.205)

3.1.1 実行を制御するコマンド

実行を制御する GDB のコマンドには次のものがあります。

- continue　　実行の再開
- finish　　関数からのリターン
- handle　　シグナル処理の変更
- jump　　実行の再開
- kill　　プログラムの削除
- remote　　コンソールの切り替え
- run　　プログラムの実行
- screen　　画面の切り替え／画面色の設定
- set args　　プログラムに渡す引数の指定
- set environment　　環境変数の設定
- show args　　プログラムの引数の表示
- step/next　　ステップ実行
- stepi/nexti　　マシンレベルステップ実行
- tty　　プログラムの標準入出力
- unset environment　　環境変数の削除
- until　　ループからの脱出

continue　実行の再開　　running

書式：　continue [<カウント>]

機能：　停止した位置から実行を再開します。

解説：　引数にはブレークポイントを無視する回数を指定することができます。

finish　関数からのリターン　　running

書式：　finish

機能：　関数からリターンします。

解説：　実行が停止している関数からリターンするまで処理を継続し，その関数からの戻り値を表示します。ただしエラーやブレークポイントがあった場合は，その位置で処理を停止します。

handle　シグナル処理の変更　running

書式：　handle <シグナル番号> <キーワード> ...

機能：　シグナルの処理方法を変更します。

解説：　<シグナル番号> に指定したシグナルが発生したときの処理を <キーワード> に指定します。

Table 3-1● キーワード

キーワード	機　能
stop	プログラムを停止する
print	メッセージを表示する
nostop	プログラムを停止させない
noprint	メッセージを表示させない
pass	プログラムがシグナルを受け取ることを許可する
nopass	プログラムがシグナルを受け取ることを許可しない

jump　実行の再開　running

書式：　jump <行番号> ········· 指定した行から実行
jump *<アドレス> ······ 指定したアドレスから実行

機能：　指定した位置から実行を再開します。

解説：　実行を再開する位置は自由に指定できますが，ブロックや関数を超えて指定しないようにしてください。

kill　プログラムの削除　running

書式：　kill

機能：　デバッグ中のプログラムを強制的に削除します。

解説：　SX-Window のプログラムの場合は，このコマンドを使用してはいけません。

remote　コンソールの切り替え　running

書式：　remote [aux | con]

機能：　**GDB** のコンソールを切り替えます[10)]。

解説：　引数には RS-232C に接続したターミナルを使用する場合は "aux" を指定します。**X680x0** に戻す場合は "con" を指定します。

10)GDB 起動時のコマンドラインオプションである '-remote' と同様の機能です。

run　プログラムの実行　running

書式：　run [<引数> ...]

機能：　デバッグ対象のプログラムをメモリにロード，実行します。

解説：　引数には，デバッグ対象プログラムのコマンドライン引数を指定することができます。指定した引数は，デバッグ対象プログラムにそのまま渡されます。
またデバッグ対象プログラムに対して，コマンドシェルのような書式でリダイレクト処理が可能です。

- **run <** 〔ファイル名〕
  標準入力を〔ファイル名〕に切り替える
- **run >** 〔ファイル名〕
  標準出力を〔ファイル名〕に切り替える
- **run >>** 〔ファイル名〕
  標準出力を〔ファイル名〕にアペンドモードで切り替える
- **run >&** 〔ファイル名〕
  標準エラー出力を〔ファイル名〕に切り替える

screen　画面の切り替え　running

書式：　screen

機能：　スクリーンスワップモードで，GDB の画面からデバッグ対象プログラムの画面へ切り替えます。

解説：　スクリーンスワップモードでプログラムが停止した場合，GDB の画面状態になりますが，このコマンドを使うことによってデバッグ対象プログラムの画面に切り替えることができます。

screen　画面色の設定　running

書式：　screen <背景の色>,<文字の色>

機能：　スクリーンスワップモードで GDB の画面の色を設定します。

解説：　引数に指定するカラーコードは 0 ～ 65535 までの値を指定します。16 進法で指定する場合は，値の前に “0x” をつけます。デフォルトでは背景を黒，文字を白に設定してあります。
次の画面は，デフォルトとは逆に背景を白，文字を黒に設定したものです (16 進法)。

```
(gdb) screen 0xffff,0
```

set args　プログラムに渡す引数の指定　running

書式：　set args [<引数> ...]

機能：　デバッグ対象プログラムのコマンドライン引数を指定します。

解説：　デバッグ対象プログラムを，次に起動するときに渡すコマンドライン引数を設定します。一度設定した引数を取り消すには，引数に何も指定せずに実行します。

set environment　環境変数の設定　running

書式：　set environment <変数名> = <値>

機能：　環境変数 <変数名> に <値> をセットします。

解説：　<値> を省略した場合は NULL が設定されます。

show args　プログラムの引数の表示　running

書式：　show args

機能：　コマンドライン引数を表示します。

解説：　"set args" コマンドで設定した，デバッグ対象プログラムに渡すコマンドライン引数を表示します。

step/next　ステップ実行　running

書式：　step [<リピート回数>]　……　関数呼び出しで関数に入る
　　　　next [<リピート回数>]　……　関数呼び出しで関数に入らない

機能：　ソースコードレベルで行単位のステップ実行をします。

解説：　<リピート回数> には実行する行数を指定することができます。省略した場合は 1 行単位でステップ実行します。

stepi/nexti マシンレベルステップ実行 running

書式： stepi [<リピート回数>] ……. 関数呼び出しで関数に入る
nexti [<リピート回数>] ……. 関数呼び出しで関数に入らない

機能： マシンレベルで行単位のステップ実行をします。

解説： <リピート回数> には実行する行数を指定することができます。省略した場合は 1 行単位でステップ実行します。

tty プログラムの標準入出力 running

書式： tty <デバイス名>

機能： デバッグ対象プログラムの標準入出力を引数に指定したデバイスに切り替えます。

解説： デバッグ対象プログラムの標準入出力を RS–232C から行いたい場合に，“tty aux” のように指定します。

unset environment 環境変数の削除 running

書式： unset environment <変数名>

機能： 環境変数を削除します。

解説： <変数名> に指定した環境変数を削除します。

until ループからの脱出 running

書式： until

機能： ループから抜け出すまで処理を継続します。

解説： プログラムカウンタがジャンプアドレスより大きくなるまで，ステップ実行を繰り返します。

3.1.2 スタックフレームを調査するコマンド

スタックフレームを調査する **GDB** のコマンドには次のものがあります。

- `backtrace/bt` **バックトレースの表示**
  `info stack/where`
- `down` **フレームの選択**
- `frame` **スタックフレームの選択と表示**
- `return` **実行の中断**
- `select-frame` **スタックフレームの選択**
- `up` **フレームの選択**

backtrace/bt/info stack/where　バックトレースの表示　　stack

書式：
backtrace ·················· スタック全体
backtrace <フレーム数> ····· 最深部から指定したフレーム数
backtrace -<フレーム数> ···· 外から指定したフレーム数
bt ························· スタック全体
bt <フレーム数> ············· 最深部から指定したフレーム数
bt -<フレーム数> ··········· 外から指定したフレーム数
info stack ················ スタック全体
info stack <フレーム数> ···· 最深部から指定したフレーム数
info stack -<フレーム数> ··· 外から指定したフレーム数
where ······················ スタック全体
where <フレーム数> ········· 最深部から指定したフレーム数
where -<フレーム数> ········ 外から指定したフレーム数

機能： デバッグ対象プログラムのスタックのバックトレースを表示します。

解説： 関数呼び出しによって生成されたスタックフレームから得ることのできる情報を表示します。バックトレースによって表示される情報は,「フレーム番号」,「関数名」,「プログラムカウンタの値」です。

down　フレームの選択　　stack

書式： down [<フレーム数>]

機能： カレントフレームの下にあるフレームを選択します。

解説： デフォルトでは <フレーム数> を 1 に設定してあります。

frame　スタックフレームの選択と表示　stack

書式： `frame` ………………… カレントフレームを選択
`frame <フレーム番号>` …… 指定したフレームを選択
`frame <アドレス>` ……… 指定したアドレスが示すフレームを選択

機能： スタックフレームを選択します。

解説： スタックフレームを選択すると同時に，そのフレームの情報から「フレーム番号」,「関数名」,「ソースファイル」,「ソースファイル上の行番号」および「その行番号の示す行のソースコード」を 1 行表示します。

return　実行の中断　stack

書式： `return`

機能： 関数の実行を途中で中断します。

解説： リターンするスタックフレームをフレームを選択するコマンドで選択しておき，このコマンドを使用することで，選択しておいたフレームに戻ることができます。

select-frame　スタックフレームの選択　stack

書式： `select-frame <フレーム番号>`
`select-frame <フレームのアドレス>`

機能： 指定したスタックフレームを選択します。

解説： <フレーム番号> は “backtrace” コマンドで調べることができます。

up　フレームの選択　stack

書式： `up [<フレーム数>]`

機能： カレントフレームの上にあるフレームを選択します。

解説： デフォルトでは <フレーム数> を 1 に設定してあります。

3.1.3 データに関するコマンド

データに関する **GDB** のコマンドには次のものがあります。

• call	**関数のテスト**
• delete display/ undisplay	**自動表示の削除**
• disable display	**自動表示の無効化**
• disassemble	**逆アセンブル**
• display	**自動表示の設定**
• enable display	**自動表示の有効化**
• output	**式の評価**
• print/inspect	**式の評価**
• printf	**フォーマットに従った値の表示**
• ptype	**型の詳細**
• set	**データのセット**
• x	**メモリの調査**
• whatis	**型の調査**

call　**関数のテスト**　　**data**

書式：　call <式>

機能：　関数をテストします。

解説：　指定した関数を **GDB** から直接実行します。関数の呼び出し引数は，プログラム内の変数または定数を指定することができます。ただし SX-Window プログラムの場合は，文字列の引数を指定しないでください。

delete display/undisplay　**自動表示の削除**　　**data**

書式：　delete display <ディスプレイ番号>
　　　　undisplay <ディスプレイ番号>

機能：　自動表示リストから削除します。

解説：　<ディスプレイ番号> は，“display” コマンドで設定したときに表示された番号です。

disable display　自動表示の無効化　data

書式：　disable display <ディスプレイ番号> ...

機能：　自動表示を無効にします。

解説：　<ディスプレイ番号> は "display" コマンドで設定したときに表示された番号です。

disassemble　逆アセンブル　data

書式：　disassemble ………………… 現在位置から実行する

disassemble <開始アドレス> ‥‥‥ 指定したアドレスから実行する

disassemble <開始> <終了> ‥‥‥ 指定範囲内で実行する

disassemble <関数名> ………… 関数の範囲内で実行する

機能：　逆アセンブルして表示します。

解説：　DOS コールおよび SX コールは，そのコールの名称で表示します。

display　自動表示の設定　data

書式：　display [/<出力フォーマット>] <式>

display [/<出力フォーマット>] <アドレス>

機能：　<式> を自動表示リストに加えます。

解説：　プログラムが停止するたびに <式> または <アドレス> を評価し表示するように設定します。引数を省略した場合は，プログラム停止時に表示された値を再表示します。

Table 3-2● 出力フォーマット

指定文字	機　能
x	16 進数で表示する
d	符号つき 10 進数で表示する
u	符号なし 10 進数で表示する
o	8 進数で表示する
c	文字定数で表示する
f	浮動小数点表記で表示する

enable display　自動表示の有効化　data

書式：　enable display <ディスプレイ番号> ...

機能：　無効にした自動表示を有効にします。

解説：　<ディスプレイ番号> は "display" コマンドで設定したときに表示される番号です。

output 式の評価 data

書式： output [/<出力フォーマット>] <式>

機能： <式> を評価し，表示します。

解説： このコマンドは "print" コマンドに似ていますが，値を表示した後に改行しません。ユーザ定義コマンドまたはコマンドファイルで使用します。

Table 3-3● 出力フォーマット

指定文字	機能
x	16 進数で表示する
d	符号つき 10 進数で表示する
u	符号なし 10 進数で表示する
o	8 進数で表示する
c	文字定数で表示する
f	浮動小数点表記で表示する

print/inspect 式の評価 data

書式： print [/<出力フォーマット>] [<式>]

inspect [/<出力フォーマット>] [<式>]

機能： <式> を評価し，その値を表示します。

解説： <式> は C 言語のスタイルで記述することができます。また **GDB** では C 言語のオペレータに加え，3 種類のバイナリオペレータを使用することができます。

Table 3-4● バイナリオペレータ

オペレータ	機能
@	メモリを配列として扱う
::	スコープ外のシンボルを参照することができる
{型}<アドレス>	指定したアドレスの値を指定した型で参照する

Table 3-5● 出力フォーマット

指定文字	機能
x	16 進数で表示する
d	符号つき 10 進数で表示する
u	符号なし 10 進数で表示する
o	8 進数で表示する
c	文字定数で表示する
f	浮動小数点表記で表示する

printf　フォーマットに従った値の表示　data

書式： printf "<フォーマット文字列>",<式>, ...

機能： <フォーマット文字列> に従って，<式> を評価した値を表示します。

解説： <フォーマット文字列> は C 言語と同様な形式で “"” と “"” で囲まれた文字列で指定します。

ptype　型の詳細　data

書式： ptype <型名>

機能： typedef によって定義されたシンボルの型を詳細に表示します。

解説： <型名> は型の名前や C 言語の “struct <タグ名>” という形式です。

set　データのセット　data

書式： set [variable] <変数名> = <値>

機能： <変数名> に <値> を代入します。

解説： デバッグ対象プログラムの変数に，<値> を代入します。
通常は，“variable” を省略できますが，GDB のサブコマンド[11]と同じ変数名の場合は省略できません。

11)‘info’ や ‘show’ コマンドのサブコマンドのことです。

x　メモリの調査　data

書式： x [/<出力フォーマット>] [<アドレス>]

機能： メモリを調査します。

解説： フォーマット指定文字で「調査サイズ」,「値の表示形式」を指定することができます。<アドレス> を指定しなければ，前に調査したアドレスの次のアドレスから調査します。

Table 3-6 ● 調査サイズ

指定文字	機　能
b	1 バイト単位で調査する
h	2 バイト単位で調査する
w	4 バイト単位で調査する
g	8 バイト単位で調査する

Table 3-7 ● 出力フォーマット

指定文字	機　能
x	符号なし 16 進数で表示する
d	符号つき 10 進数で表示する
u	符号なし 10 進数で表示する
o	符号なし 8 進数で表示する
c	文字定数で表示する[12)]
f	浮動小数点で表示する[13)]
s	NULL でターミネートされた文字列として表示する[14)]
i	逆アセンブルしてマシンインストラクションを表示する[14)]

12)キャラクタコードとして文字を表示します。

13)これは，'w' と 'g' のサイズでのみ有効です。

14)調査サイズ指定文字によって，指定されたユニットサイズは無視されます。

whatis　型の調査　　data

書式：　whatis [<シンボル>]

機能：　<シンボル> の型を表示します。

解説：　変数または関数の型を調べることができます。<シンボル> を省略した場合は，変数履歴の最後の値のデータ型を表示します。

3.1.4 ブレークポイントに関するコマンド

ブレークポイントに関する **GDB** のコマンドには次のものがあります。

● `break/rbreak`	**ブレークポイントの設定**
● `clear/ delete breakpoints`	**ブレークポイントの削除**
● `commands`	**処理の設定**
● `condition`	**ブレークポイントの停止条件の設定**
● `count`	**停止回数の設定**
● `disable`	**ブレークポイントの無効化**
● `enable`	**ブレークポイントの有効化**
● `ignore`	**パスカウントの設定**
● `tbreak`	**一時的なブレークポイントの設定**
● `watch`	**値の監視**

break/rbreak ブレークポイントの設定 breakpoints

書式：

`break` ······	カレント行
`break <行番号>` ······	ソース行
`break <ファイル名>:<行番号>` ····	指定ファイルのソース行
`break +<オフセット>` ······	カレント行から指定行後方
`break -<オフセット>` ······	カレント行から指定行前方
`break <関数名>` ······	関数のエントリー
`break <ファイル名>:<関数名>` ····	指定ファイルの関数
`break *<アドレス>` ······	アドレス
`break <位置> if <条件>` ······	条件つきブレークポイント[15]
`rbreak <文字列>` ······	<文字列> にマッチする関数すべて

15) <条件>には式を指定します。

機能： 指定した位置にブレークポイントを設定します。

解説： GDB では 1 か所に複数のブレークポイントを設定することができます。条件つきブレークポイントは，<条件> の値が 0 以外のときに停止します。

clear/delete breakpoints　ブレークポイントの削除　breakpoints

書式： clear ………………………… カレント行のブレークポイント
clear [<ファイル名>:]<位置> …… 指定位置のブレークポイント
delete <ブレークポイント番号> ‥ 番号指定
delete [breakpoints] ………… すべてのブレークポイント

機能： ブレークポイントを削除します。

解説： 指定した <位置> (行または関数) に設定されているブレークポイントを削除します。

commands　処理の設定　breakpoints

書式： commands <ブレークポイント番号>

機能： ブレークポイントで停止したときの処理を設定します。

解説： ブレークポイントで停止した後に，いくつかの処理を自動的に実行するように設定できます。

condition　ブレークポイントの停止条件の設定　breakpoints

書式： condition <ブレークポイント番号> [<条件>]

機能： ブレークポイントのコンディションを設定します。

解説： ブレークポイントの停止条件[16]を設定します。

16) <条件>は式を指定します。

count　停止回数の設定　breakpoints

書式： count <カウント>

機能： ブレークポイントに停止回数を設定します。

解説： プログラムが停止した位置のブレークポイントに<カウント>−1 に設定します。

disable　ブレークポイントの無効化　breakpoints

書式： disable <ブレークポイント番号> … 指定のブレークポイント
disable breakpoints ………… すべてのブレークポイント

機能： ブレークポイントを無効にします。

解説： 設定済みブレークポイントを一時的に無効にすることができます。

enable　ブレークポイントの有効化　breakpoints

書式： enable [breakpoints] [<モード>]<ブレークポイント番号>

機能： "disable" コマンドで無効にしたブレークポイントを有効にします。

解説： <モード>には，次にプログラムが停止するときまで一時的に有効にする "once" と，一時的に有効にしかつプログラムが停止した後，ブレークポイントを削除する "delete" を指定することができます。

ignore　パスカウントの設定　breakpoints

書式： ignore <ブレークポイント番号> <カウント>

機能： ブレークポイントを無視する回数を指定します。

解説： 次にブレークポイントに達したとき必ず停止させるには，<カウント>を 0 にしてください。

tbreak　一時的なブレークポイントの設定　breakpoints

書式：

tbreak	カレント行
tbreak <行番号>	ソース行
tbreak <ファイル名>:<行番号>	指定ファイルのソース行
tbreak +<オフセット>	カレント行から指定行後方
tbreak -<オフセット>	カレント行から指定行前方
tbreak <関数名>	関数のエントリー
tbreak <ファイル名>:<関数名>	指定ファイルの関数
tbreak *<アドレス>	アドレス

機能： 一時的なブレークポイントを設定します。

解説： このコマンドで設定したブレークポイントでプログラムが停止した後，ブレークポイントを削除します。

watch　値の監視　breakpoints

書式：

watch <シンボル>	変数
watch <アドレス>	アドレス

機能： 指定したシンボルまたはアドレスの値が変化したときに，実行を停止します。

解説： "break" コマンドと同様な方法で，削除や属性の変更が可能です。

3.1.5 ファイルに関するコマンド

ファイルに関する GDB のコマンドには次のものがあります。

• cd	**ワーキングディレクトリの設定**
• directory	**ソースファイル検索パスの設定**
• exec-file	**実行ファイルの指定**
• file	**デバッグプログラムの指定**
• list	**ソースリストの表示**
• path	**プログラム検索パスの設定**
• pwd	**ワーキングディレクトリの表示**
• reverse-search	**後方検索**
• search/ forward-search	**前方検索**
• symbol-file	**シンボル情報ファイルの指定**

cd　ワーキングディレクトリの設定　files

書式： cd <パス>
機能： カレントワーキングディレクトリを変更します。
解説： 作業用ディレクトリを変更する場合に使用します。

directory　ソースファイル検索パスの設定　files

書式： directory [<パス>[;<パス>...]]
機能： ソースファイル検索パスを設定します。
解説： <パス> は環境変数 PATH のように“;”で区切ることで複数指定することができます。

exec-file　実行ファイルの指定　files

書式： exec-file <ファイル名>
機能： 実行ファイルを指定します。
解説： “symbol-file” コマンドと一緒に使用します。

file　デバッグプログラムの指定　files

書式：　file <ファイル名>

機能：　デバッグ対象プログラムを指定します。

解説：　"exec-file" コマンドと "symbol-file" コマンドの両方を実行した場合と同じ結果になります。

list　ソースリストの表示　files

書式：

list ·························· カレント行を中心

list [<ファイル名>:]<行番号> ··· 指定行を中心

list [<ファイル名>:]<関数名> ··· 関数の始まりを中心

list [<始まり>],<終わり> ······· 指定範囲

list + ························ 最後に表示した行以降

list - ························ 最後に表示した行以前

機能：　ソースリストを表示します。

解説：　デフォルトでは 10 行ごとに表示します。

path　プログラム検索パスの設定　files

書式：　path [<パス>；...]

機能：　デバッグ対象プログラムの検索パスを設定します。

解説：　<パス> はシステムの環境変数と同様に ";" で区切ります。

pwd　ワーキングディレクトリの表示　files

書式：　pwd

機能：　カレントワーキングディレクトリを表示します。

解説：　現在の作業用ディレクトリを表示します。

reverse-search　後方検索　files

書式：　reverse-search [<文字列>]

機能：　後方向にサーチします。

解説：　ソースコードから指定した文字列を検索してその行を表示します。

search/forward-search　前方検索　files

書式：　search [<文字列>]

forward-search [<文字列>]

機能：　<文字列> を前方向にサーチします。

解説：　ソースコードから指定した文字列を検索してその行を表示します。

symbol-file　シンボル情報ファイルの指定　files

書式：　symbol-file <ファイル名>

機能：　シンボル情報のファイルを指定します。

解説：　デバッグ対象プログラムをデバッガに指定するときに使用します。

3.1.6 プログラムの状態を調査するコマンド

プログラムの状態を調査する GDB のコマンドには次のものがあります。

- `info` **プログラムの状態の調査**
- `set info/show` **GDB の状態の調査**

info　プログラムの状態の調査　status

書式： `info [<サブコマンド>] [<サブコマンドの引数> ...]`

機能： デバッグ対象プログラムの設定状態を表示します。

解説： 状態を <サブコマンド> に従ってその情報を表示します。

set info/show　GDB の状態の調査　status

書式： `set info`

`show [<サブコマンド>] [<サブコマンドの引数> ...]`

機能： GDB の設定状態を表示します。

解説： GDB の状態を <サブコマンド> に従ってその情報を表示します。

3.1.7 シンボルテーブルを調査するコマンド

シンボルテーブルを調査する **GDB** のコマンドには次のものがあります。

- `printsyms` **デバッグ情報の出力**[17)]

17)X680x0 GDB では，廃止されています。

`printsyms` デバッグ情報の出力 **obscure**

書式： `printsyms <ファイル名>`

機能： **GDB** が読み込んだシンボル情報をファイルに出力します。

解説： デバッガとデバッグ情報のデバッグに使用するコマンドです。

3.1.8 infoコマンドのサブコマンド

"info" コマンドのサブコマンドには，次のものがあります。

- `info address` **シンボルのアドレスの調査**
- `info args` **関数の呼び出し引数の調査**
- `info breakpoints` **ブレークポイントの表示**
- `info display` **自動表示リストの表示**
- `info files` **プログラム名の調査**
- `info frame` **フレームの調査**
- `info functions` **関数の調査**
- `info line` **ソース行の調査**
- `info locals` **ローカル変数の調査**
- `info registers` **レジスタの調査**
- `info signals` **シグナルの調査**
- `info source` **ソースファイルの調査**
- `info sources` **ソースファイルの調査**
- `info types` **データ型の調査**
- `info variables` **スタティックな変数の調査**
- `info watchpoints` **ウォッチポイントの表示**

`info address` シンボルのアドレスの調査 — info

書式： `info address <シンボル>`

機能： 引数に指定したシンボルのアドレスを表示します。

解説： レジスタ変数の場合は，どのレジスタに保持されているのかを表示します。ローカル変数の場合は，変数が格納されているスタックフレームのオフセットを表示します。

`info args` 関数の呼び出し引数の調査 — info

書式： `info args`

機能： 関数の引数を表示します。

解説： 実行が停止した関数の呼び出し引数をすべて表示します。

info breakpoints　ブレークポイントの表示　　info

書式：　info breakpoints

機能：　設定されているすべてのブレークポイントを表示します。

解説：　ブレークポイント番号を知ることができます。

info display　自動表示リストの表示　　info

書式：　info display

機能：　自動表示リストを表示します。

解説：　“display” コマンドで設定した自動表示リストを表示します。

info files　プログラム名の調査　　info

書式：　info files

機能：　デバッグ対象プログラムのファイル名を表示します。

解説：　シンボルを読み込むための実行ファイルの名称を表示します。

info frame　フレームの調査　　info

書式：　info frame　……………　カレントのスタックフレーム

info frame <アドレス>　….　指定アドレスのスタックフレーム

機能：　スタックフレームの情報を表示します。

解説：　指定したスタックフレームまたはカレントのスタックフレームの詳細な情報を表示します。

info functions　関数の調査　　info

書式：　info functions [<文字列>]

機能：　シンボリックデバッグ情報から，関数名とその型を表示します。

解説：　引数を省略した場合はすべての関数名を表示し，指定した場合は <文字列> にマッチする関数名を表示します。

info line　ソース行の調査　info

書式：　info line [<行番号>]

機能：　ソース行に対応するプログラムのアドレスを表示します。

解説：　引数を省略した場合は，実行が停止した行に対応するアドレスを表示します。

info locals　ローカル変数の調査　info

書式：　info locals [<文字列>]

機能：　ローカル変数をすべて表示します。

解説：　引数を省略した場合はすべての変数名とその値を，指定した場合は<文字列> にマッチする変数名とその値を表示します。

info registers　レジスタの調査　info

書式：　info registers [<レジスタ名>] [18]

機能：　レジスタを表示します。

解説：　引数には表示させたい <レジスタ名> を指定します。<レジスタ名> を指定しなければ，すべてのレジスタを表示します。

18)X680x0 GDB では，表示形式が変更されています。

info signals　シグナルの調査　info

書式：　info signals

機能：　シグナルの一覧を表示します。

解説：　シグナル番号を知ることができます。

info source　ソースファイルの調査　info

書式：　info source [<ファイル名>]

機能：　ソースコードに関する情報を表示します。

解説：　引数にファイル名を指定しなければ，現在選択されているソースファイルの情報を表示します。

info sources　ソースファイルの調査　　info

書式：　info sources

機能：　ソースファイルを表示します。

解説：　デバッグ対象プログラムのソースファイル名をすべて表示します。

info types　データ型の調査　　info

書式：　info types [<文字列>]

機能：　プログラムに定義されているデータ型を表示します。

解説：　引数を省略した場合はすべてのデータ型を，指定した場合は <文字列> にマッチするデータ型を表示します。

info variables　スタティックな変数の調査　　info

書式：　info variables [<文字列>]

機能：　extern 変数および static 変数をすべて表示します。

解説：　引数を省略した場合はすべての変数名を，指定した場合は <文字列> にマッチする変数名を表示します。

info watchpoints　ウォッチポイントの表示　　info

書式：　info watchpoints

機能：　設定されているすべてのウォッチポイントを表示します。

解説：　ウォッチポイント番号を知ることができます。

3.1.9 GDB を設定するコマンド

GDB の設定を行うコマンドには，次のものがあります。

- `define` ユーザコマンドの定義
- `document` ユーザコマンドのドキュメンテーション
- `down-silently` メッセージの抑制
- `echo` 文字列の表示
- `help` コマンドヘルプ
- `make` Make の実行
- `quit` GDB の終了
- `set` GDB の設定
- `set complaints` デバッグ情報のエラーの制御
- `set confirm` 確認の抑制
- `set editing` 行編集の設定
- `set height` スクリーンの行数の設定
- `set history filename` ヒストリの設定
- `set history save` ヒストリの設定
- `set history size` ヒストリの設定
- `set listsize` 表示リストサイズの設定
- `set print array` 表示する配列の要素数の設定
- `set print pretty` 表示する構造体の設定
- `set print union` 表示する共用体の設定
- `set prompt` プロンプトの設定
- `set radix` 基数の設定
- `set verbose` メッセージの制御
- `set width` スクリーンの桁数の設定
- `shell` チャイルドプロセスの起動
- `source` コマンドファイルの読み込み
- `up-silently` メッセージの抑制

define ユーザコマンドの定義 support

書式： `define [<コマンド名>]`

機能： コマンドを定義します。

解説： GDB のコマンドを使用して，新しくユーザがコマンドを定義することができます。

document　ユーザコマンドのドキュメンテーション　support

書式：　document [<コマンド名>]

機能：　“define” コマンドで定義したコマンドにドキュメントをつけます。

解説：　このドキュメントは “help” コマンドを使用すると表示されます。

down-silently　メッセージの抑制　support

書式：　down-silently

機能：　“down” コマンドを実行したときに GDB からのメッセージを省略します。

解説：　ユーザ定義コマンドまたはコマンドファイルの中で使用するコマンドです。

echo　文字列の表示　support

書式：　echo <文字列>

機能：　引数に指定した文字列を表示します。

解説：　ユーザ定義コマンドまたはコマンドファイルの中で使用するコマンドです。

help　コマンドヘルプ　support

書式：　help [<コマンド名>]

機能：　コマンドの説明を表示します。

解説：　引数にコマンド名を指定することで詳細な説明を表示します。引数を省略した場合は，コマンド一覧を表示します。

make　Make の実行　support

書式：　make [<引数> ...]

機能：　チャイルドプロセスで Make を実行します。

解説：　make.x が必要です。

quit　GDB の終了　support

書式： quit

機能： GDB を終了します。

解説： SX–Window のプログラムをデバッグしているときは，SX–Shell を終了してから実行してください。

set　GDB の設定　support

書式： set [<サブコマンド>] [<引数> ...]

機能： GDB の設定をします。

解説： このコマンドは，デバッグ対象プログラムのデータを設定するときと GDB の設定をするときの両方で使用します。

set complaints　デバッグ情報のエラーの制御　stauts

書式： set complaints [<無視する回数>]

機能： デバッグ情報のエラーを無視します。

解説： デバッグ情報にエラーがあり，読む込むことができない場合に設定します。引数を省略した場合は現在の状態を表示します。

set confirm　確認の抑制　support

書式： set confirm [on | off]

機能： GDB からの確認を有効または無効に設定します。

解説： 引数に “on” を指定すると，GDB はいくつかのコマンドで確認を求めるようになります。“off” を指定すると確認を求めなくなります。デフォルトは “on” です。

set editing　行編集の設定　support

書式： set editing [on | off]

機能： 行編集機能を設定します。

解説： デフォルトでは “on” に設定されています。

set height　スクリーンの行数の設定　　support

書式：　set height [<行数>]

機能：　スクリーンの行数を設定します。

解説：　引数を省略した場合は現在の値を表示します。デフォルトでは 31 行に設定されています。

set history filename　ヒストリの設定　　support

書式：　set history filename <ファイル名>

機能：　コマンドヒストリのファイル名を指定します。

解説：　デフォルトでは "./.gdb_history" になっています。

set history save　ヒストリの設定　　support

書式：　set history save [on | off]

機能：　コマンドヒストリのセーブ機能の設定をします。

解説：　"on" にした場合は，GDB を終了する際にコマンドヒストリをファイルに書き出します。デフォルトは "off" です。

set history size　ヒストリの設定　　support

書式：　set history size <サイズ>

機能：　コマンドヒストリバッファのサイズを指定します。

解説：　デフォルトは 256 バイトです。

set listsize　表示リストサイズの設定　　stauts

書式：　set listsize <行数>

機能：　"list" コマンドで表示されるソースリストの行数を設定します。

解説：　デフォルトでは 10 行に設定されています。

set print array　表示する配列の要素数の設定　support

書式：　set print array <要素数>

機能：　表示する配列の要素数を設定します。

解説：　大きな配列を表示する場合に，表示する配列を制限することができます。

set print pretty　表示する構造体の設定　support

書式：　set print pretty [on | off]

機能：　構造体の表示を見やすくします。

解説：　構造体を，1つのメンバが1行にインデントしたフォーマットで表示します。

set print union　表示する共用体の設定　support

書式：　set print union [on | off]

機能：　構造体に含まれる共用体の表示を設定します。

解説：　"on" を指定すると共用体を表示し，"off" を指定すると省略して表示します。

set prompt　プロンプトの設定　support

書式：　set prompt [<プロンプト>]

機能：　GDB のプロンプトを設定します。

解説：　デフォルトは "(gdb)" です。

set radix　基数の設定　support

書式：　set radix [2 | 8 | 10 | 16]

機能：　GDB が表示する値の基数を設定します。

解説：　デフォルトでは 10 進数に設定されています。

set symbol reloading　シンボル読み込みの制御　support

書式：　set symbol reloading [on | off]

機能：　シンボリックデバッグ情報の再読み込み機能を設定します。

解説：　シンボリックデバッグ情報は，デバッグ対象プログラム実行時に1度だけ読み込まれますが，このコマンドを“on”にすることで，プログラムの再実行時にも読み込むようになります。

set verbose　メッセージの制御　support

書式：　set verbose [on | off]

機能：　GDBからのメッセージを制御します。

解説：　GDBは，ユーザにとって必要なGDBの動作情報を表示することができます。これらのうちのいくつかは，“set verbose”コマンドで切り替えることができます。

set width　スクリーンの桁数の設定　stauts

書式：　set width [<桁数>]

機能：　スクリーンの桁数を設定します。

解説：　引数を省略した場合は現在の値を表示します。デフォルトでは96桁に設定されています。

shell　チャイルドプロセスの起動　support

書式：　shell [<コマンド>]

機能：　チャイルドプロセスを起動します。

解説：　環境変数“SHELL”に定義されているコマンドシェルを起動します。

source　コマンドファイルの読み込み　support

書式：　source [<ファイル名>]

機能：　GDBのコマンドファイルを読み込みます。

解説：　あらかじめ記述しておいたデバッグ手順のファイルを読み込む場合に使用します。

up-silently メッセージの抑制 support

書式： `up-silently`

機能： "up" コマンドを実行したときに，**GDB** からのメッセージを省略します。

解説： ユーザ定義コマンドまたはコマンドファイルの中で使用するコマンドです。

3.2 ……………………… 行編集をサポートするキー

1)テキストエディタのような行編集を可能にするプログラムです。

GDB では，キーボードからの入力を GNU readline [1] を使用しています。その GNU readline をコントロールするためのキー操作を，次に示します。

Table 3-8● キー入力一覧表

キー入力	動 作
CTRL+B	カーソルを，左に 1 文字分移動する
CTRL+F	カーソルを，右に 1 文字分移動する
DEL	カーソルを，左に 1 文字分移動し，その位置の文字を消去する
CTRL+D	カーソル位置の文字を消去する
CTRL+_	直前に行ったキー操作を取り消す
CTRL+A	カーソルを，入力した行の先頭に移動する
CTRL+E	カーソルを，入力した行の最後に移動する
ESC–F	カーソルを，1 語右に移動する
ESC–B	カーソルを，1 語左に移動する
CTRL+L	画面をクリアする
CTRL+K	カーソル位置から行の終わりまで削除する
ESC–D	カーソル位置の単語を，カーソル位置からその単語の終わりまで削除する
ESC–DEL	カーソル位置の単語を，カーソル位置の前の文字からその単語の先頭まで削除する
CTRL+W	カーソル位置からその前のスペースまで削除する
CTRL+P	直前のヒストリバッファの内容を表示する
CTRL+N	直後のヒストリバッファの内容を表示する

Chapter 4

Appendix

4.1 ……………………… 各ツールオプション一覧

4.1.1 コンパイラドライバのオプションスイッチ

- -a　ブロック単位でのプロファイラ …… 5
- -ansi　ANSI 違反の報告 …… 6
- -C　コメントを削除しない …… 7
- -c　オブジェクトファイルの生成 …… 8
- -D　マクロの定義 …… 9
- -E　プリプロセッサ処理結果の出力 …… 10
- -f　最適化の許可／禁止 …… 11
  - -fcaller-saves …… 11
  - -fcombine-regs …… 11
  - -fforce-addr …… 11
  - -fforce-mem …… 11
  - -finline-functions …… 11
  - -fkeep-inline-functions …… 12
  - -fno-defer-pop …… 12
  - -fno-function-cse …… 12
  - -fno-peep-hole …… 12
  - -fomit-frame-pointer …… 12
  - -fpcc-struct-return …… 12
  - -fstrength-reduce …… 13
  - -funsigned-char …… 13
  - -fwritable-strings …… 13
  - -ffixed-<レジスタ名> …… 13
  - -fcall-used-<レジスタ名> …… 13
  - -fcall-saved-<レジスタ名> …… 13
- -g　ソースコードデバッグ情報の生成 …… 14
- -I　インクルードパスの指定 …… 15
- -l　ライブラリの指定 …… 16
- -M　ファイル依存関係の出力 …… 16

- -MM ファイル依存関係の出力 …… 17
- -mregparm 引数をレジスタ渡しにする …… 17
- -mshort int を 16 ビットにする …… 18
- -m68881 68881 用コードの生成 …… 18
- -O 最適化の実行 …… 19
- -o 出力ファイルネームの指定 …… 20
- -p プロファイラコードの生成 …… 21
- -pedantic ANSI に厳密に適合 …… 21
- -Q バーボーズモード指定 …… 22
- -S アセンブラソースの生成 …… 23
- -traditional 伝統的な C 言語仕様に準拠 …… 24
- -trigraph trigraph シーケンスの認識 …… 24
- -U マクロの削除 …… 25
- -v コマンドラインの表示 …… 25
- -version コンパイラのバージョン表示 …… 26
- -W ワーニングの許可 …… 27
  指定されたワーニングの許可 …… 30
- -w ワーニングの禁止 …… 32

4.1.2 X68000 GCC 独自のオプションスイッチ

- -as-symbols アセンブラの最大シンボル数の指定 …… 34
- -cc1-stack コンパイラスタック量の指定 …… 34
- -cpp-stack プリプロセッサスタック量の指定 …… 35
- -fall-bsr[1] すべての関数をショートコールで指定 …… 35
- -fall-jsr[2] すべての関数をロングコールで指定 …… 36
- -fall-remote 記憶クラスの固定化 …… 36
- -fall-text テキストセクションですべてを出力する …… 37
- -fno-const-mult-expand 定数乗法展開の禁止 …… 37
- -frtl-debug rtl の書き出し …… 38
- -fscd ソースコードデバッグの生成 …… 38
- -fstrings-align 文字列の偶数整合 …… 39
- -fstack-check スタックチェックコードの生成 …… 39
- -fstrings-nopcr[3] 文字列のプログラム相対禁止 …… 40
- -fstruct-strict-align 構造体のパッキング …… 41
- -ftext-report 詳細なエラー報告 …… 42
- -fundump undump コンパイルの指定 …… 42
- -SX SX--Window プログラムモードの指定 …… 43

1)X680x0 GCC では，変更されています。

2)X680x0 GCC では，廃止されています。

3)X680x0 GCC では，廃止されています。

4.1.3 実行ファイルの条件を指定するオプションスイッチ

- -z-heap　ヒープサイズの指定 …… 44
- -z-stack　スタックサイズの指定 …… 44

4.1.4 アセンブラオプション

- -8　シンボルの識別長の指定 …… 49
- -a[4]　絶対ショートアドレス形式対応モードの指定 …… 50
- -d　全シンボルの外部定義指定 …… 51
- -f　リストファイルのフォーマット指定 …… 52
- -i　インクルードファイルのパス指定 …… 53
- -l　タイトル表示の指定 …… 54
- -m[5]　最大シンボル数の指定 …… 55
- -n　最適化の禁止 …… 56
- -o　オブジェクトファイル名の指定 …… 57
- -p　リストファイルの作成 …… 58
- -q[6]　クイックイミディエイト形式への変換禁止 …… 59
- -r[7]　相対セクション命令の使用許可 …… 60
- -s　シンボルの定義 …… 61
- -t　テンポラリファイルのパス指定 …… 62
- -u　未定義シンボルの外部参照指定 …… 63
- -w　ワーニングレベルの指定 …… 64
- -x　シンボル情報の出力指定 …… 66
- -z　**HAS** オリジナル機能へのワーニング禁止 …… 67

4)X680x0 HAS では，廃止されています。

5)X680x0 HAS では，変更されています。

6)X680x0 HAS では，廃止されています。

7)X680x0 HAS では，廃止されています。

4.1.5 リンカオプション

- -a　実行ファイルの拡張子の省略時に‘.x’をつけない …… 69
- -d　外部定義シンボルの登録 …… 70
- -i　インダイレクトファイルの指定 …… 71
- -l　ライブラリパスの使用 …… 72
- -m　最大シンボル数の指定 …… 73
- -o[8]　実行ファイル名の指定 …… 74
- -p　マップファイルの作成 …… 75
- -s　OBJR 形式の情報の作成 …… 76
- -t　タイトル表示の指定 …… 77
- -v　バーボーズモードの指定 …… 78

8)X680x0 HLK では，バグフィックスしています。

- -w ワーニングメッセージの抑制 79
- -x シンボルテーブルの出力禁止 80
- -z -v オプションを無効にする 81

4.1.6 デバッガオプション

- -batch バッチ処理 83
- -cd ワーキングディレクトリの指定 84
- -command コマンドファイルの指定 85
- -directory ソースディレクトリのパス指定 86
- -exec[9] デバッグ対象プログラムの指定 87
- -epoch[10] Emacs を使用する 88
- -fullname[11] Emacs を使用する 88
- -help ヘルプメッセージの表示 89
- -mem メモリの設定 90
- -nx 初期化ファイルを読み込まない 91
- -quiet タイトルの非表示 92
- -remote リモートコンソールモードで起動する 93
- -r リモートコンソールモードで起動する 93
- -se[12] デバッグ対象プログラムの指定 94
- -symbols[13] シンボル情報のファイルの指定 95
- -swap スクリーンスワップモードで起動する 96
- -tty 標準入出力先の指定 97

9)X680x0 GDB では，廃止されています。

10)X680x0 GDB では，廃止されています。

11)X680x0 GDB では，廃止されています。

12)X680x0 GDB では，廃止されています。

13)X680x0 GDB では，廃止されています。

4.2 ……………………………… GDB コマンド一覧

4.2.1 実行を制御するコマンド

- continue　実行の再開 …… 181
- finish　関数からのリターン …… 181
- handle　シグナル処理の変更 …… 182
- jump　実行の再開 …… 182
- kill　プログラムの削除 …… 182
- next　ステップ実行 …… 184
- nexti　マシンレベルステップ実行 …… 185
- remote　コンソールの切り替え …… 182
- run　プログラムの実行 …… 183
- screen　画面の切り替え／画面色の設定 …… 183
- set args　プログラムに渡す引数の指定 …… 184
- set environment　環境変数の設定 …… 184
- show args　プログラムの引数の表示 …… 184
- step　ステップ実行 …… 184
- stepi　マシンレベルステップ実行 …… 185
- tty　プログラムの標準入出力 …… 185
- unset environment　環境変数の削除 …… 185
- until　ループからの脱出 …… 185

4.2.2 スタックフレームを調査するコマンド

- backtrace　バックトレースの表示 …… 186
- bt　バックトレースの表示 …… 186
- down　フレームの選択 …… 186
- frame　スタックフレームの選択と表示 …… 187
- info stack　バックトレースの表示 …… 186

- **return** 実行の中断 187
- **select-frame** スタックフレームの選択 187
- **up** フレームの選択 187
- **where** バックトレースの表示 186

4.2.3 データに関するコマンド

- **call** 関数のテスト 188
- **delete display** 自動表示の削除 188
- **disable display** 自動表示の無効化 189
- **disassemble** 逆アセンブル 189
- **display** 自動表示の設定 189
- **enable display** 自動表示の有効化 189
- **inspect** 式の評価 190
- **output** 式の評価 190
- **print** 式の評価 190
- **printf** フォーマットに従った値の表示 191
- **ptype** 型の詳細 191
- **set** データのセット 191
- **undisplay** 自動表示の削除 188
- **x** メモリの調査 191
- **whatis** 型の調査 192

4.2.4 ブレークポイントに関するコマンド

- **break** ブレークポイントの設定 193
- **clear** ブレークポイントの削除 194
- **commands** 処理の設定 194
- **condition** ブレークポイントの停止条件の設定 194
- **count** 停止回数の設定 194
- **delete breakpoints** ブレークポイントの削除 194
- **disable** ブレークポイントの無効化 194
- **enable** ブレークポイントの有効化 195
- **ignore** パスカウントの設定 195
- **rbreak** ブレークポイントの設定 193
- **tbreak** 一時的なブレークポイントの設定 195
- **watch** 値の監視 195

4.2.5 ファイルに関するコマンド

- `cd` ワーキングディレクトリの設定 …… 196
- `directory` ソースファイル検索パスの設定 …… 196
- `exec-file` 実行ファイルの指定 …… 196
- `file` デバッグプログラムの指定 …… 197
- `forward-search` 前方検索 …… 198
- `list` ソースリストの表示 …… 197
- `path` プログラム検索パスの設定 …… 197
- `pwd` ワーキングディレクトリの表示 …… 197
- `reverse-search` 後方検索 …… 197
- `search` 前方検索 …… 198
- `symbol-file` シンボル情報ファイルの指定 …… 198

4.2.6 プログラムの状態を調査するコマンド

プログラムの状態を調査する **GDB** のコマンドには次のものがあります。

- `info` プログラムの状態の調査 …… 199
- `set info` **GDB** の状態の調査 …… 199
- `show` **GDB** の状態の調査 …… 199

4.2.7 シンボルテーブルを調査するコマンド

- `printsyms`[1] デバッグ情報の出力 …… 200

1)X680x0 GDB では廃止されています。

4.2.8 info コマンドのサブコマンド

- `info address` シンボルのアドレスの調査 …… 201
- `info args` 関数の呼び出し引数の調査 …… 201
- `info breakpoints` ブレークポイントの表示 …… 202
- `info display` 自動表示リストの表示 …… 202
- `info files` プログラム名の調査 …… 202
- `info frame` フレームの調査 …… 202
- `info functions` 関数の調査 …… 202
- `info line` ソース行の調査 …… 203
- `info locals` ローカル変数の調査 …… 203

- info registers　レジスタの調査 ………… 203
- info signals　シグナルの調査 ………… 203
- info source　ソースファイルの調査 ………… 203
- info sources　ソースファイルの調査 ………… 204
- info types　データ型の調査 ………… 204
- info variables　スタティックな変数の調査 ………… 204
- info watchpoints　ウォッチポイントの表示 ………… 204

4.2.9 GDB を設定するコマンド

- define　ユーザコマンドの定義 ………… 205
- document　ユーザコマンドのドキュメンテーション ………… 206
- down-silently　メッセージの抑制 ………… 206
- echo　文字列の表示 ………… 206
- help　コマンドヘルプ ………… 206
- make　Make の実行 ………… 206
- quit　GDB の終了 ………… 207
- set　GDB の設定 ………… 207
- set complaints　デバッグ情報のエラーの制御 ………… 207
- set confirm　確認の抑制 ………… 207
- set editing　行編集の設定 ………… 207
- set height　スクリーンの行数の設定 ………… 208
- set history filename　ヒストリの設定 ………… 208
- set history save　ヒストリの設定 ………… 208
- set history size　ヒストリの設定 ………… 208
- set listsize　表示リストサイズの設定 ………… 208
- set print array　表示する配列の要素数の設定 ………… 209
- set print pretty　表示する構造体の設定 ………… 209
- set print union　表示する共用体の設定 ………… 209
- set prompt　プロンプトの設定 ………… 209
- set radix　基数の設定 ………… 209
- set symbol reloading　シンボル読み込みの制御 ………… 210
- set verbose　メッセージの制御 ………… 210
- set width　スクリーンの桁数の設定 ………… 210
- shell　チャイルドプロセスの起動 ………… 210
- source　コマンドファイルの読み込み ………… 210
- up-silently　メッセージの抑制 ………… 211

4.3 ……………………………… 診断メッセージ一覧

4.3.1 GCC エラーメッセージ

- \x の後が１６進表現ではありません …… 142
- '{ }' 表現は関数内部だけで使用できます …… 153
- 'a5' は使えません …… 157
- *Action* は異なったタイプです …… 152
- bit-field '*Ident*' に不当な type があります …… 116
- bit-field '*Ident*' のサイズが 0 です …… 116
- bit-field '*Ident*' の幅が整数ではありません …… 115
- bit-field に 'sizeof' は適用できません …… 135
- bit-field のメンバ '*Ident*' のアドレスは参照できません …… 150
- break が loop か switch の中にありません …… 139
- case の値が同値です …… 137
- case ラベルが switch 文の中にありません …… 136
- case ラベルが整数型ではありません …… 136
- char '*Ident*' に long, short 指定されています …… 106
- '*CHAR*' が不正な位置にあります …… 155
- char 配列を広い文字で初期化しています …… 130
- continue が loop 外部です …… 139
- default ラベルが switch 文の中にありません …… 137
- default ラベルが２つ以上あります …… 138
- Dump ファイル書出しに失敗しました …… 154
- Dump ファイルを作れません *Filename* …… 154
- 'enum *Ident*' が再宣言されています …… 117
- enum '*Ident*' の値が整数定数ではありません …… 118
- extern'*Ident*' は初期化できません …… 126
- 'f' が２つ以上あります …… 141
- field'*Ident*' が関数に宣言されています …… 112
- field'*Ident*' が不完全です …… 113

- frame pointer にできないタイプです 156
- global register 変数は関数定義の後に宣言できません 157
- global register 変数は初期化できません 156
- '*Ident*' が built-in 関数と衝突しました 101
- '*Ident*' が void の配列です 109
- '*Ident*' が関数の配列です 109
- '*Ident*' が再宣言されました 100
- '*Ident*' が宣言の前に使われました 102
- '*Ident*' が複数のデータタイプを持っています 105
- '*Ident*' が別記憶クラス宣言されました 100
- '*Ident*' が別の宣言をされました 100
- *Ident* がレジスタ変数ではありません 156
- '*Ident*' と衝突しています 101
- '*Ident*' に long と short 双方指定されています 106
- '*Ident*' に signed と unsigned 双方指定されています 107
- '*Ident*' の long, short, signed , unsigned は不正です 106
- '*Ident*' の記憶クラスが複数です 107
- '*Ident*' のサイズが定数ではありません 104
- '*Ident*' のサイズは不明です 104
- *Ident* の初期値のサイズが決定できません 128
- '*Ident*' の引数が少なすぎます 149
- '*Ident*' の引数が多すぎます 147
- '*Ident*' の前宣言位置です 102
- '*Ident*' の要素が不完全なタイプです 128
- '*Ident*' は register 宣言できないタイプです 156
- '*Ident*' は struct か union のはずです 144
- '*Ident*' は typedef か built in type できません 105
- '*Ident*' は違法なタグです 103
- '*Ident*' は不完全です 125
- '*Ident*' は未宣言です 125
- int 配列を短い文字で初期化しています 130
- 'l' が2つ以上あります 141
- 'l' が3つ以上あります 141
- label *Label* がありません 103
- LASCII 文字列同士は連結できません 140
- LASCII 文字列には wide 文字は使えません 154
- LASCII 文字列は 255 文字までです 154
- operand は最大8個です 155
- '*Operate*' は不正です 145
- output オペランドに '=' がありません 155
- output オペランドに不正な '+' があります 155

- relocate は関数内部で宣言できません 157
- relocate は初期化できません 157
- struct *Ident* が再定義されました 115
- struct に '*Ident*' がありません 143
- switch 文の条件が整数ではありません 153
- SXCALL 関数のアドレスはとれません 154
- SXCALL 関数は inline にできません 114
- top-level での 'auto' は不正です 108
- typedef '*Ident*' が変数のように初期化されています 127
- 'u' が 2 つ以上あります 141
- union *Ident* が再定義されました 115
- union に '*Ident*' がありません 143
- union に初期化すべきメンバがありません 131
- void に値はありません 135
- 違法な'\' です 140
- 演算 '*Operator*' オペランドが不正です 150
- オペランドが複数存在しています 155
- オペランドが矛盾しています 155
- 可変サイズオブジェクトは初期化できません 132
- 空の宣言です 124
- 関数 '*Ident*' const で volatile な関数は違法です 110
- 関数 '*Ident*' が変数のように初期化されています 127
- 関数 '*Ident*' 関数を返すことはできません 111
- 関数 '*Ident*' の記憶クラスが不正です 113
- 関数 '*Ident*' 配列を返すことはできません 111
- 関数 '*Ident*' は定義できません 155
- 関数 '*Ident*' はプロトタイプ必須です 111
- 関数でないオブジェクトを呼ぼうとしています 147
- 関数の記憶クラスが 'auto' です 108
- 関数の記憶クラスが 'common' です 108
- 関数の記憶クラスが 'register' です 108
- 関数の記憶クラスが 'relocate' です 108
- 関数の記憶クラスが 'remote' です 108
- 関数の記憶クラスが 'typedef' です 108
- 関数の外で可変サイズ変数は使えません 155
- 関数の引数が多すぎます 148
- 構造体には変換できません 134
- 最低追加必要量 *Num* バイトです 156
- 左辺値ではありません 142
- 条件式での type が異なっています 151
- 省略引数は空の引数宣言とはマッチしません 101

- 初期化式が定数ではありません 129
- 初期化式が複雑すぎます 130
- 初期化式に { } は不正です 131
- 初期化要素は 1 つしか必要ありません 131
- 数字表現が違法です 140
- 数字末尾が違法です 141
- スタックが不足です 156
- 整数型に変換できません 133
- 添え字が整数ではありません 146
- 添え字を持つ変数が配列かポインタではありません 146
- 配列 ‘*Ident*’ のサイズが整数ではありません 109
- 配列 ‘*Ident*’ のサイズが負です 110
- 配列 ‘*Ident*’ の要素数がありません 129
- 配列参照に添え字がありません 145
- 配列にキャストできません 151
- 配列範囲が未定義です 125
- 配列を非定数表現で初期化しています 130
- 引数 ‘*Ident*’ がありません 121
- 引数 ‘*Ident*’ がプロトタイプと異なります 122
- 引数 ‘*Ident*’ が初期化されています 129
- 引数 ‘*Ident*’ が不完全です 114
- 引数 ‘*Ident*’ が不完全です 121
- 引数 ‘*Ident*’ が複数あります 120
- 引数 ‘*Ident*’ は void と宣言されています 120
- 引数が 2 つのスタイルで渡されています 119
- 引数がありません 119
- 引数が少なすぎます 149
- 引数の void は 1 つしか存在できません 115
- 引数の数がプロトタイプと異なります 122
- 引数の名前がありません 120
- 引数は double のみマッチします[1] 124
- 引数は int のみマッチします[2] 123
- 不正な #pragma dump です 154
- 不正な void 表現です 142
- 不正な初期化です 132
- 不正なレジスタ ‘*RegName*’ がレジスタ変数に指定されています 156
- 不適切な数字表現です 140
- 不適切な浮動小数表現です 140
- 浮動小数点型にできません 133
- 浮動小数点表現が違法です 140
- 不当な表現です 150

1)X680x0 GCC では，発生しません。

2)X680x0 GCC では，発生しません。

3)X680x0 GCC では，発生しません。

4)X680x0 GCC では，発生しません。

- フレームポインタレジスタが不正です 156
- プロトタイプは double です[3] 124
- プロトタイプは int です[4] 123
- 文法違反 140
- 変数‘*Ident*’が void に宣言されました 112
- 変数‘*Ident*’は初期化できません 127
- 変数‘*Ident*’を SXCALL クラス指定しています 114
- ポインタスケールが不完全です 144
- ポインタ宣言が不正です 112
- ポインタに変換できません 133
- ポインタは浮動小数点型にできません 133
- メンバ‘*Ident*’が2重です 117
- 文字がありません 141
- 文字列定数が長すぎます 141
- 文字列表現が誤っています 141
- 戻り値が不完全です 118
- 呼び出し関数の引数が不完全です 148
- ラベル‘*Ident*’が複数あります 102
- レジスタ変数‘*Ident*’は 68881 指定が必要です 156
- 列挙型が整数範囲を越えています 118

4.3.2 GCC ワーニングメッセージ

- ++は適用できません 166
- ‘,’が enum list の終わりにあります 162
- −−は適用できません 166
- ‘;’が struct か union に余計にあります 162
- ‘;’が struct か union の終わりにありません 162
- \x の後が16進表現ではありません 163
- 16進表現が範囲を越えています 164
- *Action* で const ポインタをコピーしています 167
- *Action* で volatile ポインタをコピーしています 167
- *Action* でポインタから整数にしました 167
- *Action* で整数からポインタにしました 167
- *Action* は異種のポインタです 167
- ANSI では const,volatile 関数は使えません 160
- ANSI C では‘{}’表現は使用出来ません 170
- ANSI C では const や volatile である関数は違法です 160
- ANSI C では constructor 表記はできません 170
- ANSI C では enum の前方参照はできません 161

- ANSI C では long long integer は違法です 170
- ANSI C では register 名指定変数は使えません 171
- ANSI C では可変サイズ配列 '*Ident*' は使えません 160
- ANSI C では空の {} での初期化は違法です 163
- ANSI C では漢字は識別子に使えません 170
- ANSI C では関数の外の ';' は違法です 163
- ANSI C ではサイズ 0 の配列 '*Ident*' は使えません 159
- ANSI C ではポインタと関数へのポインタを比較できません 165
- ANSI C ではメンバのない宣言は違法です 162
- ANSI C は '?　:' 表現を省略できません 170
- ANSI C は newline で文字定数を連結できません 164
- ANSI C は void *と関数へのポインタとの条件式は許していません 165
- ANSI C は void *と関数ポインタの比較を許しません 165
- ANSI C は構造体を構造体にキャストすることは許していません 166
- ANSI C は非左辺値配列の添え字を許しません 165
- ANSI C はポインタと関数を比較することを許していません 165
- bit-field '*Ident*' の幅は負にできません 161
- bit-field '*Ident*' は ANSI C では不当です 161
- case 値 'Value' は enum'*Ident*' に存在していません 168
- enum が内部宣言されました 161
- global register が関数内で再使用されています 171
- '*Ident*' が前に int を返すと暗黙に宣言されています 158
- '*Ident*' が暗黙に宣言されました 159
- '*Ident*' が初期化されずに使用されているようです 168
- '*Ident*' が宣言されて定義されませんでした 169
- '*Ident*' が複数あります 159
- '*Ident*' が未使用です 168
- *Ident* タグ名 '*Ident*' が引数として宣言されました 161
- '*Ident*' の宣言は local 変数を隠します 158
- '*Ident*' の宣言は引数を隠します 158
- '*Ident*' は 'int' です 162
- '*Ident*' は 'longjmp' で破壊される可能性があります 169
- '*Ident*' は暗黙に関数として宣言されています 158
- '*Ident*' は定義されて使用されませんでした 169
- '*Ident*' は未使用の引き数です 169
- '*Ident*' はリードオンリーです 166
- '*Ident*' を 'extern' で初期化しようとしています 159
- '*Ident0*' と '*Ident1*' は非常に類似した識別子です 169
- label *Label* が参照されていません 162
- long long long は長すぎます 159
- non-void 関数が値を返していません 163

- sizeof が void type に使われています ······ 164
- sizeof が関数に使われています ······ 164
- static'*Ident*' が暗黙に extern に宣言されています ······ 157
- struct,union でない初期化式に '{}' があります ······ 167
- switch で列挙値 '*Ident*' の処理がありません ······ 168
- type or storage class がありません ······ 162
- union が内部宣言です ······ 161
- union のメンバがありません ······ 161
- void である関数に値を持った return があります ······ 168
- void でない関数に値のない return があります ······ 167
- void ポインタを減算に用いています ······ 165
- void ポインタを算術演算しています ······ 165
- volatile 関数は戻ってきません ······ 163
- volatile 宣言された関数に return があります ······ 167
- ＸＣ拡張表現です ······ 170
- 意味のない keyword,type name が宣言にあります ······ 159
- 意味のない文です ······ 168
- 異種ポインタを比較しています ······ 165
- 可変長引数関数は inline にできません ······ 160
- 可変長引数関数は inline にできません ······ 171
- 漢字を 'L' なく定数に使っています ······ 170
- 関数 'main' は inline にできません ······ 160
- 関数が値を返す場合、そうでない場合があります ······ 163
- 関数で破壊されるレジスタが global レジスタ変数に使われています ······ 171
- 関数へのポインタを減算に用いています ······ 165
- 関数へのポインタを算術演算しています ······ 165
- 関数宣言が prototype ではありません ······ 160
- キャストなしで異なるポインタを比較しました ······ 165
- 局所宣言 '*Ident*' が外部と一致しません ······ 158
- 結果が使用されていません ······ 168
- コード最適化は行われていません ······ 171
- 条件式が何時も 0 です ······ 166
- 条件式が何時も 1 です ······ 166
- 条件式で定数を代入しています ······ 166
- 条件式で論理演算結果を代入しています ······ 166
- 条件式に pointer/integer 双方存在します ······ 166
- 条件式のポインタが異種です ······ 165
- 初期化要素が多すぎます ······ 167
- スコープは宣言か定義の中だけです ······ 161
- 整数定数が範囲を越えました ······ 164
- タイプの指定のない引数の名前が関数宣言に現れました ······ 160

- 多文字文字定数です ………… 164
- 二重の const です ………… 159
- 二重の volatile です ………… 159
- 変換タイプが 'const' です ………… 167
- 変換タイプが 'volatile' です ………… 167
- 変数 '*Ident*' が inline 宣言です ………… 160
- ポインタと整数を比較しています ………… 165
- ポインタを整数 0 と大小比較しています ………… 165
- 未知のエスケープシーケンスです ………… 164
- 未知のレジスタです: *RegName* ………… 171
- 戻り値は int です ………… 162
- レジスタ変数 '*Ident*' のアドレスを求めています ………… 166
- 割り込み関数は inline にできません ………… 171

4.3.3 アセンブラのメッセージ

- Abort: Out of memory ………… 172
- Abort: Device full ………… 172
- Abort: Too many symbols[5] ………… 172
- *file name* file open error ………… 172
- temporary file open error ………… 172
- Forced error by fail directive ………… 173
- bad opecode error ………… 173
- division by zero error ………… 173
- expression error ………… 173
- feature not available error ………… 173
- file not found error ………… 173
- illegal addressing error ………… 173
- illegal operand error ………… 173
- illegal quick size error ………… 173
- illegal relative error ………… 173
- illegal shift count error ………… 173
- illegal size error ………… 173
- illegal symbol error ………… 173
- illegal value error ………… 173
- macro nesting over error ………… 173
- missing if error ………… 173
- missing macro error ………… 173
- no symbol error ………… 173
- overflow error ………… 174

5) X680x0 HAS では，発生しません。

- redefinition error ······ 174
- register error ······ 174
- register size error ······ 174
- too many include file error ······ 174
- undefined symbol error ······ 174
- Warning: illegal register list ······ 174
- Warning: terminator not found ······ 174
- Warning: illegal alignment ······ 174
- Warning: HAS expanded specifications[6] ······ 174
- Warning: short addressing ······ 175
- Warning: illegal short value ······ 175
- Warning: absolute addressing ······ 175
- Warning: absolute short addressing ······ 175

6)X680x0 HASでは，発生しません。

4.3.4 HLKのメッセージ

- Duplicate definition : *symbol name* ······ 176
- Already read : *file name* ······ 176
- Bad option : *option name* ······ 176
- Calc stack over flow in *xxxx* ······ 176
- Calc stack under flow in *xxxx* ······ 176
- Can't open file : *file name* ······ 176
- Device full : *file name* ······ 176
- Division by zero in *xxxx* ······ 176
- File I/O error : *file name* ······ 176
- Illegal expression in *xxxx* ······ 176
- Illegal file size : *file name* ······ 176
- Illegal SCD information in *xxxx* ······ 177
- Indirect mode error ······ 177
- Internal error at : *xxxx* ······ 177
- Mismatch roffset size (?_?) ! ······ 177
- Not found : *file name* ······ 177
- Not found indirect file : *file name* ······ 177
- Not obj, arc file : *file name* ······ 177
- Out of memory !! (°_°; ······ 177
- Over flow in *xxxx* ······ 177
- Relative error in *xxxx* ······ 177
- Too many arguments ······ 177
- Undefined environment variable 'lib' ······ 177
- Undefined symbol(s) in *file name* ······ 177

- **Unknown option :** *option name* ········· 178
- **Unknown command :** *xxxx* ········· 178
- **Warning, duplicate definition :** *symbol name* ········· 178

4.4 ……………………… アセンブラ擬似命令一覧

4.4.1 アセンブラ制御

- .comment　コメント行の指定 ………………………… Vol.1 99
- .cpu　アセンブラ対象命令セットの指定 ………… Vol.1 100
- .end　プログラムの終了指定 …………………… Vol.1 98
- .fail　エラーの生成 …………………………… Vol.1 100
- .include　ソースコードの挿入 ……………………… Vol.1 98
- .org　ロケーションカウンタの指定 …………… Vol.1 99
- .request　リンク時のライブラリ指定 ……………… Vol.1 98

4.4.2 セクション指定

- .bss　ブロックストレージセクションの宣言 ……… Vol.1 102
- .comm　コモンエリアの指定 ………………………… Vol.1 103
- .data　データセクションの宣言 ………………… Vol.1 101
- .offset　オフセットテーブルの定義開始 ………… Vol.1 102
- .rbss　相対セクションの宣言 ……………………… Vol.1 104
- .rcomm　相対コモンエリアの指定 ………………… Vol.1 104
- .rdata　相対セクションの宣言 ……………………… Vol.1 104
- .rlbss　相対セクションの宣言 ……………………… Vol.1 104
- .rlcomm　相対コモンエリアの指定 ………………… Vol.1 104
- .rldata　相対セクションの宣言 ……………………… Vol.1 104
- .rlstack　相対セクションの宣言 ……………………… Vol.1 104
- .rstack　相対セクションの宣言 ……………………… Vol.1 104
- .stack　スタックセクションの宣言 ……………… Vol.1 102
- .text　テキストセクションの宣言 ……………… Vol.1 101

4.4.3 外部名の宣言

- .entry 外部定義名の宣言 Vol.1 107
- .extrn 外部参照名の宣言 Vol.1 107
- .external 外部参照名の宣言 Vol.1 107
- .global グローバルシンボルの宣言 Vol.1 106
- .globl グローバルシンボルの宣言 Vol.1 106
- .public 外部定義名の宣言 Vol.1 107
- .xdef 外部定義名の宣言 Vol.1 107
- .xref 外部参照名の宣言 Vol.1 107

4.4.4 シンボルの定義

- = 可変シンボル値の定義 Vol.1 108
- .equ 不変シンボル値の定義 Vol.1 108
- .reg レジスタリストの定義 Vol.1 109
- .set 可変シンボル値の定義 Vol.1 108

4.4.5 マクロ制御

- .endm マクロ定義の終了 Vol.1 110
- .exitm マクロ展開の打ち切り Vol.1 111
- .irp 不定回数の繰り返しの開始 Vol.1 113
- .irpc 不定回数の文字繰り返しの開始 Vol.1 113
- .local マクロ定義ブロック内の局所的シンボルの定義 Vol.1 111
- .macro マクロ定義の開始 Vol.1 110
- .rept 繰り返しの開始 Vol.1 112

4.4.6 データの定義

- .dc 定数データの定義 Vol.1 114
- .dcb 定数ブロックの定義 Vol.1 115
- .ds メモリ領域の確保 Vol.1 115
- .even 偶数境界の調整 Vol.1 116

4.4.7 条件つきアセンブル

- .else 反対の条件でアセンブル実行 ················ Vol.1 117
- .elseif 反対の条件が成立しかつ指定の条件が真のときにアセンブル実行 ···························· Vol.1 117
- .endc 条件つきアセンブルの終了 ·················· Vol.1 118
- .endif 条件つきアセンブルの終了 ·················· Vol.1 118
- .if 条件が真のときにアセンブル実行 ············· Vol.1 117
- .ifdef シンボルが定義されているときにアセンブル実行 Vol.1 117
- .ifeq 条件が偽のときにアセンブル実行 ············· Vol.1 117
- .iff 条件が偽のときにアセンブル実行 ············· Vol.1 117
- .ifndef シンボルが定義されていないときにアセンブル実行 ···························· Vol.1 117
- .ifne 条件が真のときにアセンブル実行 ············· Vol.1 117

4.4.8 リストファイル制御

- .lall マクロ展開行の出力 ························· Vol.1 122
- .list アセンブルリストの出力 ···················· Vol.1 119
- .nlist アセンブルリストの出力抑制 ················ Vol.1 119
- .page アセンブルリストの改ページ/ページ長指定 ···· Vol.1 120
- .sall マクロ展開行の出力抑制 ···················· Vol.1 123
- .subttl アセンブルリストのサブタイトル指定 ········· Vol.1 121
- .title アセンブルリストのタイトル指定 ············ Vol.1 121
- .width アセンブルリストの表示桁数の指定 ··········· Vol.1 120

4.4.9 シンボリックデバッグ情報の指定

- .def～.endef シンボルテーブルエントリの作成 ············ Vol.1 125
- .dim 配列の指定 ································· Vol.1 130
- .file ソースファイル名の出力指定 ················ Vol.1 124
- .line 行番号の指定 ······························ Vol.1 129
- .ln 行番号とロケーションの対応の出力指定 ······ Vol.1 124
- .scl 記憶クラスの宣言 ························· Vol.1 126
- .size サイズの指定 ······························ Vol.1 129
- .tag タグ名の宣言 ······························ Vol.1 128
- .type C 言語における型の宣言 ···················· Vol.1 127
- .val シンボルの値の指定 ························ Vol.1 126

SOFT BANK ソフトバンク

ISBN4-89052-533-5
C0055 P5300E
9784890525331
2冊セット定価5,300円
(セット本体5,146円　分売不可)
1910055053007

X68k
Programming Series
(#1)
X680x0
Develop.